AN OBJECT ORIENTED DESIGN AND IMPLEMENTATION OF THE IDEF$_0$ ESSENTIAL DATA MODEL USING ADA AND AN ADA BASED EXPERT SYSTEM

THESIS

Terry LeVere Kitchen
Captain, USAF

AFIT/GCS/ENG/90D-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/90D-07

AN OBJECT ORIENTED DESIGN AND IMPLEMENTATION OF THE IDEF$_0$ ESSENTIAL DATA MODEL USING ADA AND AN ADA BASED EXPERT SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

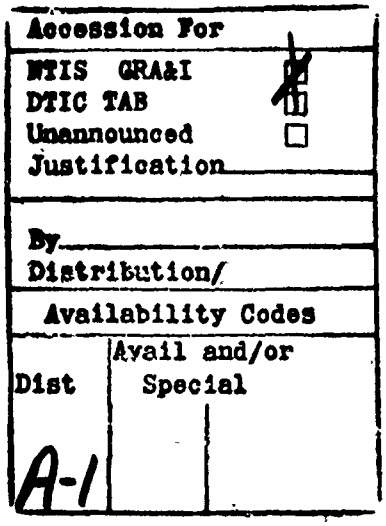
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)



Terry LeVere Kitchen, B.S.C.S.

Captain, USAF

DECEMBER, 1990

Preface

This investigation develops and implements an Ada based object oriented design (OOD) of the $IDEF_0$ Essential Data Model called the Essential Subsystem, which includes an Ada based expert system for $IDEF_0$ model syntax checking. $IDEF_0$ is a graphic approach to system description developed by SofTech, Inc. for the U.S. Air Force Program for Integrated Computer-Aided Manufacturing (ICAM) and is a subset of the Structured Analysis (SA) language (30, 35). The $IDEF_0$ Essential Data Model is an entity-relationship (E-R) model of the $IDEF_0$ language and represents the fundamental (essential) information of an $IDEF_0$ model.

The development of the Essential Subsystem, as well as SAtool II, is part of ongoing research at the Air Force Institute of Technology, in association with the Strategic Defense Initiative Organization (SDIO), on the use of $IDEF_0$ as a software requirements modeling methodology. This research is performed to demonstrate the feasibility of Ada and object oriented design techniques in the development of CASE tools and the use of Ada in expert systems, but the primary objective was to provide a subsystem that will be integrated with SAtool II.

I would like to thank the many people who supported me during this research. I begin by thanking Capt Jay-Evan Tevis with whom much of this research has been performed. I thank my thesis co-advisor, Capt Robert Hammell, for his always forthright advice and guidance throughout this research. I would also like to thank my thesis co-advisor, Dr. Gary Lamont, for providing additional meaning to the terms learning and professionalism. In addition, I thank the other member of my committee, Dr. Thomas Hartrum who assumed the role of customer and provided many spirited discussions on the $IDEF_0$ language. Finally, my greatest thanks go to my wife, Kathy, whose love, devotion, and moral support kept me going through all the long days and nights.

Terry LeVere Kitchen

Table of Contents

The following thesis volume is maintained at the Air Force Institute of Technology, Department of Electrical and Computer Engineering. The points of contact are Dr. Gary B. Lamont or Dr. Thomas C. Hartrum.

Volume II: Essential Subsystem Source Code

List of Figures

List of Tables

Abstract

This ~~investigation~~ thesis develops and implements an Ada based object oriented design (OOD) of the IDEF$_0$ Essential Data Model called the Essential Subsystem, which includes an Ada based expert system for IDEF$_0$ model syntax checking. IDEF$_0$ is a graphic approach to system description developed by SofTech, Inc. for the U.S. Air Force Program for Integrated Computer Aided Manufacturing (ICAM) and is a subset of the Structured Analysis (SA) language (30, 36). The IDEF$_0$ Essential Data Model is an entity-relationship (E-R) model of the IDEF$_0$ language and represents the fundamental (essential) information of an IDEF$_0$ model. The Essential Subsystem is so named because of its intended use as a subsystem for integration with the Ada based, IDEF$_0$ CASE tool, SAtool II, that is under concurrent development ~~(40)~~. The development of SAtool II is part of ongoing research at the Air Force Institute of Technology (AFIT), with the Strategic Defense Initiative Organization (SDIO), on the use of IDEF$_0$ as a software requirements modeling methodology.

The IDEF$_0$ Essential Data Model and its corresponding data dictionary representation were created during an earlier AFIT research effort and are revised as part of the requirements analysis phase of this investigation. The E-R constructs of the IDEF$_0$ Essential Data Model are then mapped into objects in an OOD. This OOD is then augmented with the necessary objects for storing and retrieving the state of IDEF$_0$ models. The design of the Essential Subsystem is completed by modeling the interface to the Ada expert system as another object in the OOD. The OOD is then implemented in Ada, including the expert system which is implemented using CLIPS/Ada (an Ada version of CLIPS). Thus, the feasibility of Ada and object oriented design techniques in the development of CASE tools and the use of Ada in expert systems is demonstrated.

The results of this investigation provide the foundation for future research into syntactical validation, automatic generation, and application specific expert modeling of $IDEF_0$ models as well as providing a subsystem for integration with SAtool II.

AN OBJECT ORIENTED DESIGN AND IMPLEMENTATION OF THE $IDEF_0$ ESSENTIAL DATA MODEL USING ADA AND AN ADA BASED EXPERT SYSTEM

I. INTRODUCTION

General Issues

Steadily rising Department of Defense (DOD) software development and maintenance costs have forced the DOD to look for cost reduction techniques in the software development life cycle. Two DOD mandates on the use of the programming language Ada in software development projects have been used to combat the problem (12, 13). The integration of Computer Aided Software Engineering (CASE) tools with expert systems has been found to be another method for improving software development life cycle efficiency and thus lowering costs (29:114). Therefore, the Air Force can probably reduce its software development and maintenance costs by research and subsequent implementation of these integrated tools. Furthermore, in addition to making integration easier, even greater savings may be realized if DOD mandates are followed, and CASE tools and expert systems are implemented in Ada.

This research covers two major areas: the development and implementation of an Object Oriented Design (OOD) for an Ada based CASE tool, and the development of an Ada based expert system for integration with that CASE tool. The preceding statement of the problem is both broad and incomplete but is restated more clearly after some additional background to the problem is provided.

Background

Before examining the foundation for this investigation, some terms must be defined. These terms provide background information that is necessary to correctly interpret prior research efforts.

Computer Aided Software Engineering (CASE). Prior to the late 1970s, the most common method for representing user requirements for system development was narrative English (43:123). These requirements exhibited several undesirable characteristics (43:123–124):

- They were monolithic.
- They were redundant.
- They were ambiguous.
- They were difficult to maintain.

The recognized need for an improved methodology led to the gradual formalization of earlier methods into methods that were graphic, partitioned, and minimally redundant (43:124–125). Early formalized methods included Data Flow Diagrams (DFDs), Entity-Relationship (E-R) Diagrams, DeMarco Data Structure Diagrams, Jackson Data Structure Diagrams, and Structured Analysis (SA) (36) (43:299–300). However, without automated tools to draw and validate the graphical models, the process of developing and maintaining the models sometimes became overwhelming, especially for systems whose requirements constantly changed. Naturally, this spurred research and development of a class of products known as Computer Aided Software Engineering (CASE) tools which automated the drawing and validation process. Therefore, by the middle 1980s, a CASE industry developed, offering several dozens of automated tools (43:128,464).

Structured Analysis (SA). A similar yet alternative graphical modeling method to the DFD is SA developed by Ross (36) (43:299). According to Ross, the SA technique produces:

> a hierarchically organized structure of separate diagrams, each of which exposes only a limited part of the subject to view, so that even very complex subjects can be understood. The structured collection of diagrams is called a *SA Model.* (36:17)

SA permits requirements and high-level design to be modeled in one of two ways: data decomposition or activity (process) decomposition (36:19). SA is the basis for the development of the Structured Analysis Design Technique (SADT[1]) by SofTech, Inc. The SADT is in wide use in Europe, the Far East, and U.S. aerospace manufacturing (28:2). The SADT is described in the book *SADT: Structured Analysis and Design Technique* by Marca and McGowan [2] (28).

$IDEF_0$. $IDEF_0$ is a requirements modeling technique developed by SofTech for the U.S. Air Force program for Integrated Computer-Aided Manufacturing (ICAM) (30). In fact, $IDEF_0$ stands for ICAM Definition Method Zero. $IDEF_0$ defines a subset of SA that omits the data decomposition and only permits requirements to be functionally modeled. The purpose of $IDEF_0$ is the "representation of the functions of a manufacturing system or environment" (30:1-1). For example, $IDEF_0$ is used to "standardize the description of aerospace manufacturing across government contractors" (28:133). However, $IDEF_0$ can be used not only to describe existing systems but also systems not yet developed. Thus, $IDEF_0$ can be used as a graphical language for modeling system requirements, including software systems, and it is in this context that it is discussed in this research.

Data Dictionary. A data dictionary is a modeling technique that usually accompanies one of the graphical modeling techniques. Its purpose is reflected by its name – providing a dictionary of the data. It usually consists of a list of data item names, and accompanying each name is a description. Names that represent composite objects are normally accompanied by a list of the data items of which it is composed (39:82–83).

[1]SADT is a registered trademark of SofTech, Inc.
[2]The description of SADT in this text is very similar to the IDEF0 subset of SA.

SAtool. During the past several years, AFIT has been actively involved in developing a CASE tool for assisting the software engineer in the requirements phase of the software development life cycle. Specifically, a 1987 MS Thesis by Steven Johnson developed the CASE tool called SAtool, which was written in the C programming language and developed for the SUN-3 workstation (19). SAtool's graphical language is based on $IDEF_0$ which, in turn, is based on the SADT. SAtool allows the user to perform requirements analysis by developing $IDEF_0$ diagrams and associated data dictionaries (19:6-1).

Expert Systems. In the late 60s and early 70s, expert systems became recognized as a separate field of study within the larger field of Artificial Intelligence (9:157). Expert systems are, in fact, computer programs. However, they exhibit several features which together distinguish them from conventional programs (9:151):

- They reason with domain-specific knowledge.
- They use domain-specific algorithms.
- They perform well in the context of a specific problem area.
- They possess an "explanation" facility, which permits the output of both the knowledge base (i.e., the domain-specific knowledge) and the logical reasoning process (i.e., the search process) in human readable terms.
- They possess a "knowledge acquisition" facility, which permits greater flexibility in modifying the domain-specific knowledge.

One of the fundamental design differences between expert systems and conventional programs of today is the separation of the domain-specific knowledge from the program that reasons with the knowledge (9:161). Thus, the design of an expert system typically consists of a knowledge base component (the domain-specific knowledge) and an inference engine component (the reasoning or search programs).

Expert System Tools. An expert system tool is any computer language or programming system that supports the eliciting and encoding of domain knowledge and provides one or more inference techniques to apply the knowledge in order to solve the problem at hand (9:175–176). For example, the programming language C is not an expert system tool, because it does not include a "built-in" inference technique. However, CLIPS (C Language Integrated Production System) is an expert system tool, because it includes an inference engine and a knowledge acquisition methodology.

Additional Background

The Air Force Institute of Technology, Department of Electrical and Computer Engineering (AFIT/ENG), has sponsored continuing MS thesis research on the integration of SAtool with an expert system. The purpose of integrating an expert system with SAtool is to provide syntax checking of the $IDEF_0$ diagram that is created by SAtool. The expert system thus frees the developer from the tedious and time consuming task of validating that his or her $IDEF_0$ diagram is free of $IDEF_0$ language violations.

In 1988, the MS thesis work of Jung began the process of integrating SAtool with an expert system by starting with a small prototype (20). SAtool is modified to accommodate the expert system by adding an option to the SAtool user interface which translates the structural representation of the $IDEF_0$ diagram into a file of expert system facts. This fact file is then transferred from the SUN-3 to a Z-248 microcomputer where it becomes part of the knowledge base for an expert system. The expert system is written in Prolog-1 and consists of rules defining the proper representation of an $IDEF_0$ diagram (20:4-2—4-4). Although the expert system is successful in performing the syntax validation of the $IDEF_0$ diagram, the rules only check a limited number of $IDEF_0$ features. In addition, full integration with SAtool is not achieved, since the fact file must be transferred to a separate computer, the Z-248.

In 1990, the MS thesis work of Kim continued research on the integration of an expert system with SAtool (24). The rule base of the expert system is extended to include rules for several additional features of the $IDEF_0$ language. The need for the separate microcomputer to run the expert system is also eliminated. The expert system is implemented in Quintus Prolog on the SUN-3 -- the same hardware platform as SAtool. However, fully transparent integration is still not achieved due to software compatibility problems between the C language and the Quintus version of Prolog (24:6-2). The entire process of $IDEF_0$ diagram creation, editing, and error checking is performed on the SUN-3, but the user is required to run two separate processes: one for SAtool and one for Quintus Prolog.

Concurrent with the MS thesis work of Jung and Kim was the research of Neelon Smith (38). The development of an Ada based version of SAtool called SAtool II in an X-Windows environment is the focus for the research. The requirements document used is an abstract data model of the $IDEF_0$ language. The abstract data model consists of two parts: an essential data model and a drawing data model. The research goal is to develop an object based Ada CASE tool (SAtool II) using the abstract data model as the requirements document (38:4-1). The development and implementation of an object oriented design (OOD) in Ada for the essential data model is achieved, but an OOD is neither designed nor implemented for the drawing data model (38:6-1). Furthermore, the OOD for the essential data model is considered inadequate because of its poor design and implementation characteristics.

Concurrent with this thesis investigation is the MS thesis research of Jay Tevis (40). Tevis is to implement SAtool II in an X-Windows environment. To that end, the research concentrates on the design and implementation of an OOD for both the drawing data model and a graphical user interface. Therefore, to get a complete picture of the creation of the CASE tool SAtool II, it is necessary to refer to (40) as well as this research.

Problem Statement

An Ada based version of SAtool (SAtool II) is still required to demonstrate the feasibility of the use of Ada and object oriented design techniques in the development of CASE tools. The development and integration of an Ada based expert system with SAtool II to perform syntactical checks of $IDEF_0$ models is also required to demonstrate the feasibility of using Ada in expert system development for application in the software development process.

For this research, the problem to be solved is modeled as the development of a subsystem that will eventually be integrated with SAtool II[3]. This subsystem consists of two parts which must be integrated together: an essential data model component and an expert system component.

Assumptions

In order to constrain the scope of the investigation, several assumptions were made at the outset of this research.

1. One or more Ada based expert system tools is available for evaluation.
2. Concurrent research work with the drawing data model and related SAtool II implementation issues (40) must proceed at a pace that does not hinder this research.
3. Users and/or researchers planning to utilize this work, must be familiar with the concepts of modeling software requirements using $IDEF_0$, SA, or SADT.

Standards

Code and documentation developed for this thesis adhere to AFIT's *System Development Documentation Guidelines and Standards (Draft # 4)* (16) whenever possible.

[3]The other components necessary for implementation of the tool are developed by Tevis (40).

Research Approach

This investigation is performed in several phases. The following paragraphs explain each of the phases.

In phase one of this research, a prototype that demonstrates the feasibility of integrating a client Ada program with an Ada based expert system is performed. A transparent integration is the prime concern. The Ada expert system chosen for this prototype is an Ada version of CLIPS (CLIPS/Ada). It is selected because of its immediate availability and its satisfaction of the expert system requirements specification.

The second phase involves a review of the requirements for SAtool II and the expert system. The SAtool II requirements include the $IDEF_0$ Essential Data Model and the corresponding data dictionary formats. Modifications to the requirements are made where inconsistencies or inadequacies are discovered.

In the third phase, an OOD for the subsystem is developed. The objects from the essential data model are identified by developing and applying a mapping technique between Entity-Relationship (E-R) Models and objects in an object oriented design. Additional objects in the OOD are derived from other SAtool II requirements and the expert system requirements.

The fourth phase concerns the actual implementation of the OOD for the subsystem. This includes the development of several utility programs to translate the $IDEF_0$ information contained in essential data model data structures (i.e., $IDEF_0$ diagram information) into facts. These facts are suitable for storage in either the working memory of the expert system or in an ASCII file for later retrieval.

At this point, it should be noted that phases two, three and four are reaccomplishments of a significant portion of earlier research (38). Unfortunately, both the object oriented design and implementation of the earlier research are considered to have design and implementation errors which render them unsuitable for incorporation into this research.

The fifth phase involves the creation of the Ada expert system to perform $IDEF_0$ syntax checking. Actually, only the development of the rule base remains, because CLIPS/Ada includes an inference engine, and the facts are obtained via the results of the fourth phase. Thus, a set of $IDEF_0$ syntax rules are developed.

The final phase involves the integration of the subsystem with the rest of SAtool II that is under concurrent development (40).

Equipment

The target environment for SAtool II development and implementation is the SUN workstation running a version of Berkeley Unix. Several workstations are readily available within the Department of Electrical and Computer Engineering to accomplish this research. The SUN is the chosen platform, because it is the most readily available workstation with the required Ada compiler and X-Windows capability within the department. However, alternative hardware platforms are possible if they support both Ada and X-Windows. It should be noted that the subsystem created in this research contains no "hooks" to X-Windows and therefore should be able to execute on any machine with a validated Ada compiler and sufficient memory.

Scope and Limitations

This investigation is limited to the areas described below:

1. A joint review of the SAtool II requirements (40) which does not include a validation of those requirements.
2. The development and implementation of an Ada based OOD for the essential data model.
3. The development and implementation of an Ada based expert system to perform syntactical checks of $IDEF_0$ models.

4. The integration of the OOD for the essential data model and the expert system into a single subsystem.
5. The development of one or more subprograms within the subsystem to translate information stored in the essential data model data structures into facts suitable for loading into the working memory of an expert system or for storing into an ASCII file.
6. The integration of the subsystem with SAtool II.

Sequence of Presentation

This thesis is organized into six chapters. Chapter Two presents a short introduction to the $IDEF_0$ language, AFIT's work with $IDEF_0$, mapping E-R models to an OOD, expert systems, and the integration of expert systems with CASE tools.

The third chapter presents a review of the requirements for the subsystem. Specifically, the essential data model, the data dictionaries, and the expert system requirements are reviewed.

The fourth chapter presents the object oriented design of the subsystem, and chapter five presents information concerning the implementation, testing, and integration of the subsystem with SAtool II.

Finally, the sixth chapter presents a summary of the investigation and some conclusions and recommendations.

Readership

This research is intended for those readers with interests in the use of Ada, expert systems, entity-relationship diagrams, and object oriented design techniques in the software development process. Furthermore, readers with interest in CASE, SA, SADT, or $IDEF_0$ in software requirements modeling should consider examining this research as well.

II. LITERATURE REVIEW

Introduction

The objective of this research investigation is to create a subsystem for the CASE tool SAtool II. A major part of the subsystem is the development and implementation of an OOD for the essential data model portion of the $IDEF_0$ abstract data model. Another part of the subsystem is the development and implementation of an Ada based expert system to perform syntactical checks of the $IDEF_0$ models created by SAtool II. The subsystem is then to be integrated with SAtool II.

Since the $IDEF_0$ language (30) is implemented by SAtool II, an overview of the language is presented first. Some modifications to the $IDEF_0$ language made by AFIT (16, 15), and implemented by SAtool II, are presented next. An abstract data model of the $IDEF_0$ language (3) is then presented. Because the abstract data model is used as a basis for deriving a mapping technique to objects in an OOD, information on the mapping of E-R constructs to objects in an OOD is also included. Finally, both expert systems (27) and their integration with CASE tools (20, 24, 38, 41, 42) are presented, since the OOD for the essential data model must be integrated with an expert system.

$IDEF_0$

As discussed in Chapter One, $IDEF_0$ is a graphical language whose purpose is the modeling of manufacturing systems, but it is also used to model software system requirements. An $IDEF_0$ model consists of several constructs which are all cross-referenced to each other (30:3–1):

1. diagrams (composed of boxes, arrows, and text)
2. texts (the facing-page text, for example)
3. glossary (performing a function similar to a data dictionary)

An $IDEF_0$ model of software system requirements is constructed by starting with an A-0 diagram that consists of a single box and a number of arrows. This diagram is a high level abstract view of the entire software system. In fact, it is conceptually similar to the context diagram which is part of the DFD modeling technique (43:339). Since the A-0 diagram lacks the necessary detail to describe the entire system, it must be decomposed into lower level diagrams forming a hierarchy, where each lower level in the hierarchy reveals greater detail. Therefore, each diagram in the model, with the exception of the A-0 diagram, is essentially a decomposition of a box in a higher level diagram. The box in the higher level diagram is appropriately called the *parent box* of the diagram. Figure 1 illustrates the hierarchical structure of an $IDEF_0$ model.

Within an individual diagram, a box represents an *activity* the system performs, whereas an arrow represents a *thing* or data that is processed by the system. Arrows in $IDEF_0$, unlike arrows in DFDs, do not represent flow or sequence but represent constraints (30:3–1). The four types of arrows used by $IDEF_0$ are as follows:

1. An *input arrow* enters the left side of a box and represents input data that may be needed in order for that activity represented by the box to be performed.
2. A *control arrow* enters the top of a box and represents control information that determines how an activity is performed.
3. An *output arrow* exits from the right side of a box and represents data that is output by the activity.
4. A *mechanism arrow* enters the bottom of a box and represents either a person or device used to carry out the activity.

Arrows, however, are not as simple as the above definitions imply. A special type of mechanism arrow that exits from the bottom of a box is named a *call*. The call arrow indicates that the function performed by the activity can be found in a decomposition in another $IDEF_0$ model. In

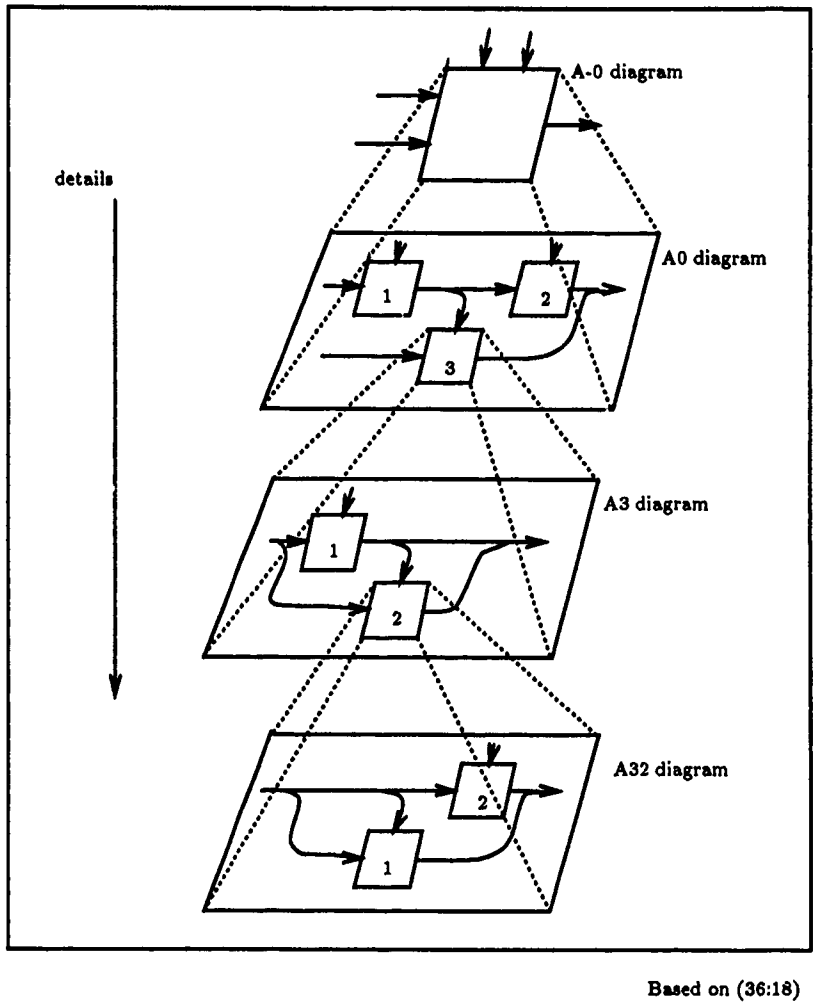


Figure 1. IDEF$_0$ Structured Decomposition

fact, a call arrow is the only arrow which does not represent a *thing* or *data*. Furthermore, each and every box in an IDEF$_0$ diagram must have at least one control arrow and one output arrow. No restrictions exist on the number of input or mechanism arrows permitted. The IDEF$_0$ manual provides additional insight on the distinction between input and control arrows:

> The assumption is that "an arrow is a control unless it obviously serves only as an input" (30:3-4).

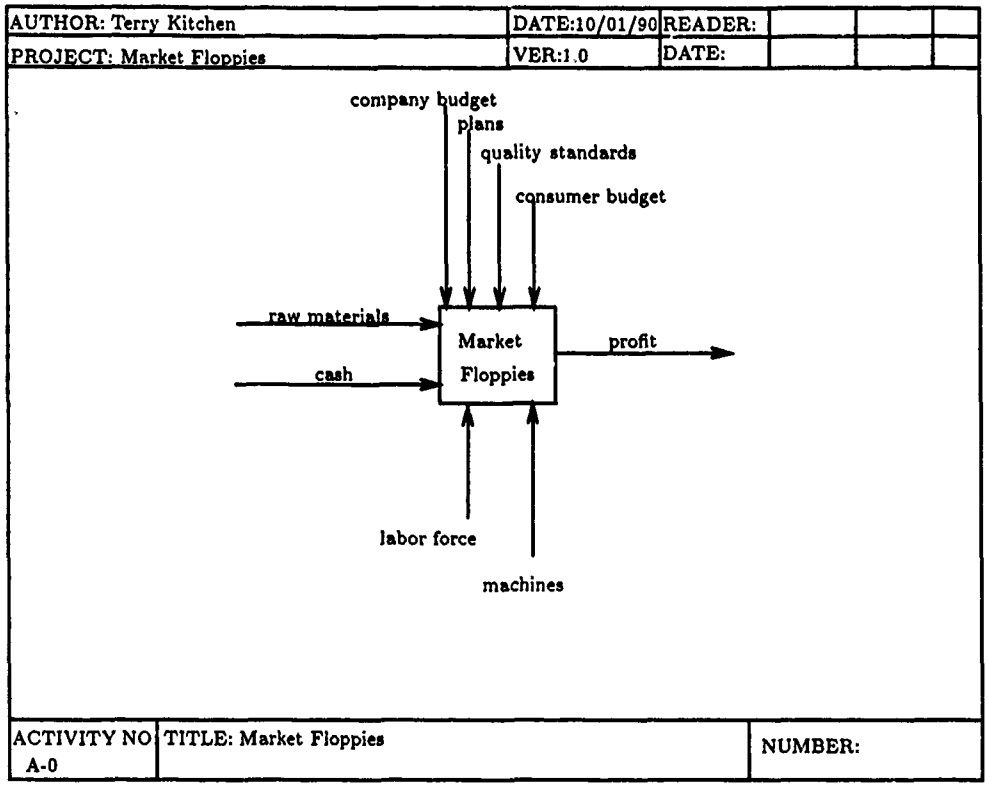


Figure 2. Sample $IDEF_0$ Diagram

Figure 2 illustrates a single $IDEF_0$ diagram that represents a contrived project for a company that markets floppy disks. In this case, it is the A-0 diagram, "Market Floppies". The arrows on the diagram fall into the following categories: 'cash' and 'raw materials' are input arrows; 'company budget', 'consumer budget', 'quality standards', and 'plans' are control arrows; 'profit' is an output arrow; and 'labor force' and 'machines' are mechanism arrows. For a more detailed account of the $IDEF_0$ modeling technique, refer to the $IDEF_0$ manual (30).

AFIT and $IDEF_0$

$IDEF_0$ Modifications. AFIT/ENG has adopted $IDEF_0$ as its requirements analysis methodology and, in doing so, has made several modifications to the language (15, 16). The following list highlights some of the modifications:

1. The $IDEF_0$ language requires an accompanying glossary with each model, but AFIT has extended the $IDEF_0$ language to include a data dictionary which, in effect, replaces the glossary (3, 16).

2. The $IDEF_0$ language states that an arrow represents some data or a *thing*, but AFIT uses the term *data element* instead.

3. The $IDEF_0$ language requires that each diagram (except the A-0 diagram) contain between three and six boxes, but AFIT has not adopted this requirement.

AFIT requires a data dictionary entry be made for each activity and data element that appears in an $IDEF_0$ model (15, 16). By replacing the $IDEF_0$ glossary of unspecified format with data dictionary entries of specific format, the syntactical precision of the $IDEF_0$ model is increased and syntactical ambiguities are reduced (3:641). Figure 3 illustrates the required data dictionary entry format for an activity, and Figure 4 illustrates the required data dictionary entry format for a data element. The 'S', 'M' an 'G' classifications that precede the fields in the data dictionary entry are based on the notation in (11:109) and refer to words single-line field, multiple-line field, and group-field respectively:

- (S) means that there is only one field of this type in the entry, and it appears on a single line.
- (M) means that there is only one field of this type in the entry, but the field consists of multiple lines.
- (G) means two or more fields are grouped together and multiple groups are allowed. However, each group member is still a single field that can only appear on a single line.

It should be noted that the 'M' field classification is inappropriately applied to some fields. For example, the DESCRIPTION field of either the activity or data element data dictionary entry is a single field that can consist of multiple lines, so the 'M' classification is correct. However,

(S) NAME:	(activity name)	C25
(S) TYPE:	(defaults to ACTIVITY)	N/A
(S) PROJECT:	(project name)	C12
(S) NUMBER:	(Node number of this activity)	C20
(M) DESCRIPTION:	(text description)	C60
(M) INPUTS:	(data element name)	C25
(M) OUTPUTS:	(data element name)	C25
(M) CONTROLS:	(data element name)	C25
(M) MECHANISMS:	(data element name)	C25
(G) ALIASES:	(an activity name that's an alias)	C25
COMMENT:	(why the alias is needed)	C60
(S) PARENT ACTIVITY:	(activity name of parent)	C25
(S) REFERENCE:	(cite a reference to the activity)	C60
(S) REF TYPE:	(the type of the reference)	C25
(S) VERSION:	(version of this entry)	C10
(S) VERSION CHANGES:	(what's different about this version)	C60
(S) DATE:	(mm/dd/yy of this entry)	C8
(S) AUTHOR:	(author's name)	C20

based on (16:14) and (11:112)

Figure 3. Data Dictionary Entry Format for Activity

the INPUTS field for the activity data dictionary entry is actually multiple fields, where each field appears on a different line. This classification is not implied by 'M'.

An Abstract Data Model for IDEF$_0$. Over the past several years AFIT has developed many CASE tools to assist students in the software development process. When developing software of any kind (including CASE tools), ambiguities in the requirements will likely lead to user dissatisfaction. A clear, concise, and "complete" statement of the requirements is essential. Therefore, one could say a "formal model" of the requirements is necessary.

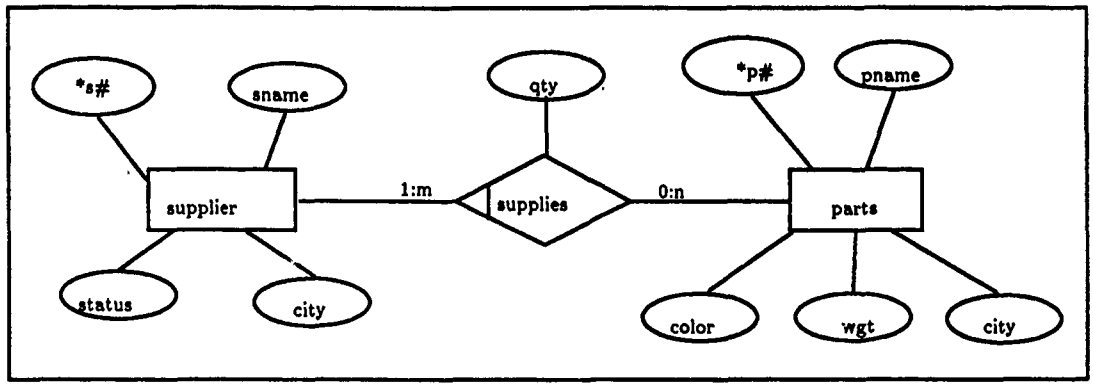
No formal model existed for the IDEF$_0$ language, and the lack of such a model increased the difficulty in developing CASE tools designed to manipulate the IDEF$_0$ language. Therefore, in 1989 a group of AFIT students and faculty developed an abstract data model of the IDEF$_0$ language (3).

The modeling technique chosen to construct the abstract data model is an extended version of the entity-relationship (E-R) modeling technique (10). Figure 5 is a sample of the technique

(S) NAME:	(data element name)	C25
(S) TYPE:	(defaults to DATA ELEMENT)	N/A
(S) PROJECT:	(project name)	C12
(M) DESCRIPTION:	(text description)	C60
(S) DATA TYPE:	(the type of data, if known)	C15
(S) MIN VALUE:	(minimum data value, if known)	C15
(S) MAX VALUE:	(maximum data value, if known)	C15
(S) RANGE:	(range of values, if applicable)	C60
(M) VALUES:	(allowable values, if appropriate)	C15
(S) PART OF:	(parent data element name)	C25
(M) COMPOSITION:	(subcomponent data element names)	C25
(G) ALIASES:	(an alias data element name)	C25
WHERE USED:	(where does the alias occur?)	C60
COMMENT:	(why the alias is needed)	C60
(M) SOURCES:	(activity name)	C25
DESTINATIONS:		N/A
(M)INPUT:	(activity(s) where it is an input)	C25
(M)CONTROL:	(activity(s) where it is a control)	C25
(S) REFERENCE:	(cite a reference to the data element)	C60
(S) REF TYPE:	(the type of the reference)	C25
(S) VERSION:	(version of this data element)	C10
(S) VERSION CHANGES:	(what's different about this version)	C60
(S) DATE:	(mm/dd/yy of this entry)	C8
(S) AUTHOR:	(author's name)	C20

based on (16:16) and (11:114)

Figure 4. Data Dictionary Entry Format for Data Element



(31:37)

Figure 5. Modified Entity Relationship Diagram

applied to a contrived example. The E-R model itself consists of rectangles (entities), ellipses (attributes), diamonds (relationships between entities), and lines (links). The extensions include either a horizontal or vertical line through one side of the diamond indicating from which direction the relationship should be read. The cardinality of the relationships is added to both sides of the diamond. Finally, an asterisk is assigned to the attribute which is the primary key.

The abstract data model consists of two parts: an essential data model and a drawing data model (3:2). The essential data model contains the information fundamental (essential) to the $IDEF_0$ model. On the other hand, the drawing data model contains information pertaining to the graphical constructs of a diagram.

> The basic premise in our approach to modeling $IDEF_0$ syntax is that a given $IDEF_0$ decomposition is actually just a graphic representation of a more fundamental underlying model, which could equally well be represented by a number of alternate diagrams without altering the analysts fundamental model. These alternatives could vary simply by one activity box being in a slightly different position, or by one using an $IDEF_0$ shorthand notation, such as a double headed arrow or a footnote. To support this approach, then, two models were developed: the ***essential model***, which is the underlying fundamental model, and the ***drawing model*** which defines one of the possibly many graphical representations of the essential model (3:642).

For example, if a model contains an activity called *compile*, the box representing compile may have

many different physical locations (coordinates) on a diagram, and yet it would still represent the same activity (compile). Therefore, the coordinates of the box, like other graphical characteristics of a diagram, are not considered essential to the fundamental $IDEF_0$ model and are placed in the drawing data model.

The primary entities of the essential data model thus become activities and data elements, while the primary entities in the drawing model are their graphical counterparts – boxes and line segments. Because of the size and complexity of both the essential and drawing data models, each is split into two parts. Figure 6 illustrates the part of the essential data model that details an activity. Figure 7 illustrates the part of the essential data model that details a data element. Figures 8 and 9 illustrate the parts of the drawing data models for an activity (box) and data element (line segment) respectively.

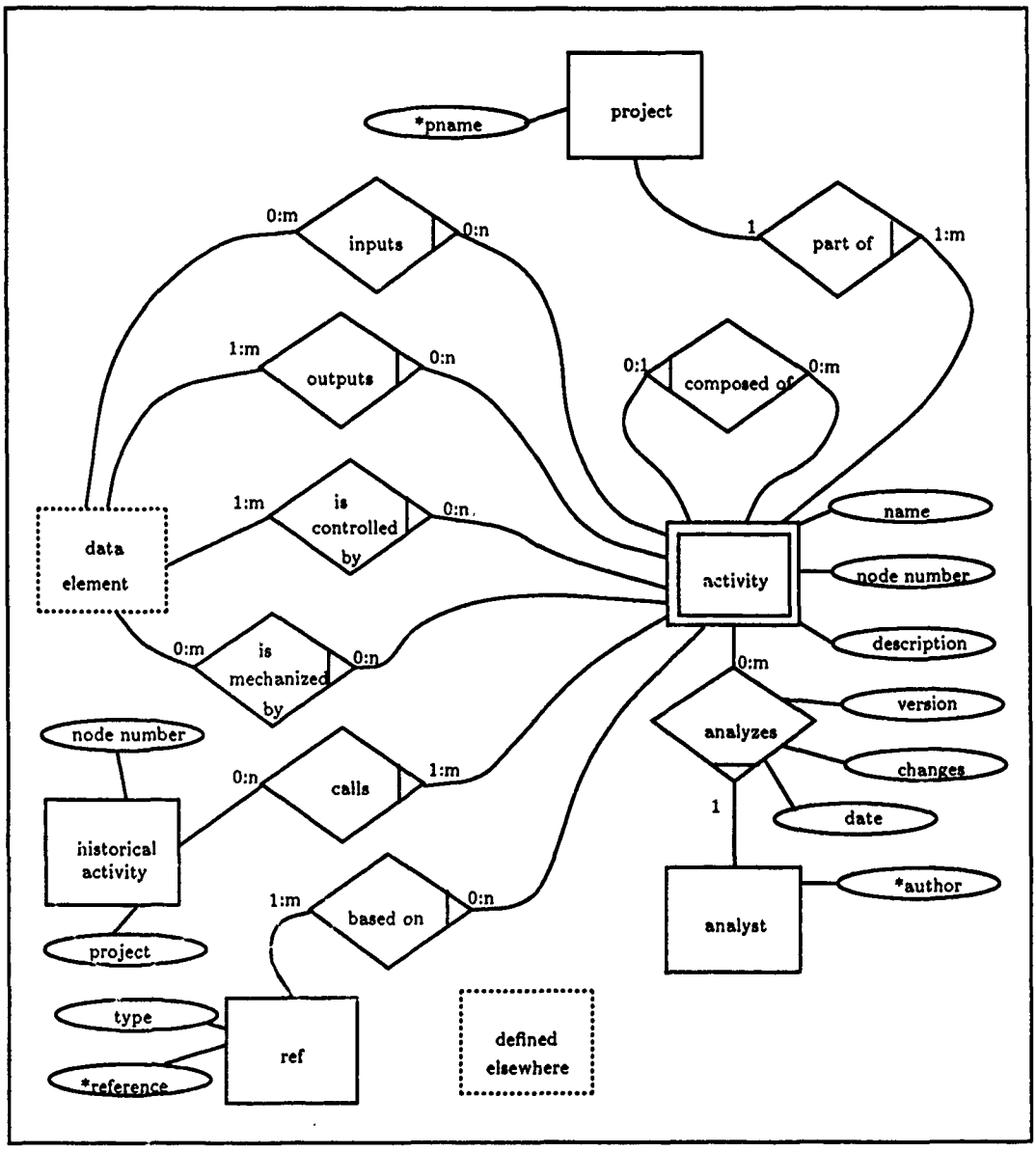


Figure 6. IDEF$_0$ Activity Essential Data Model

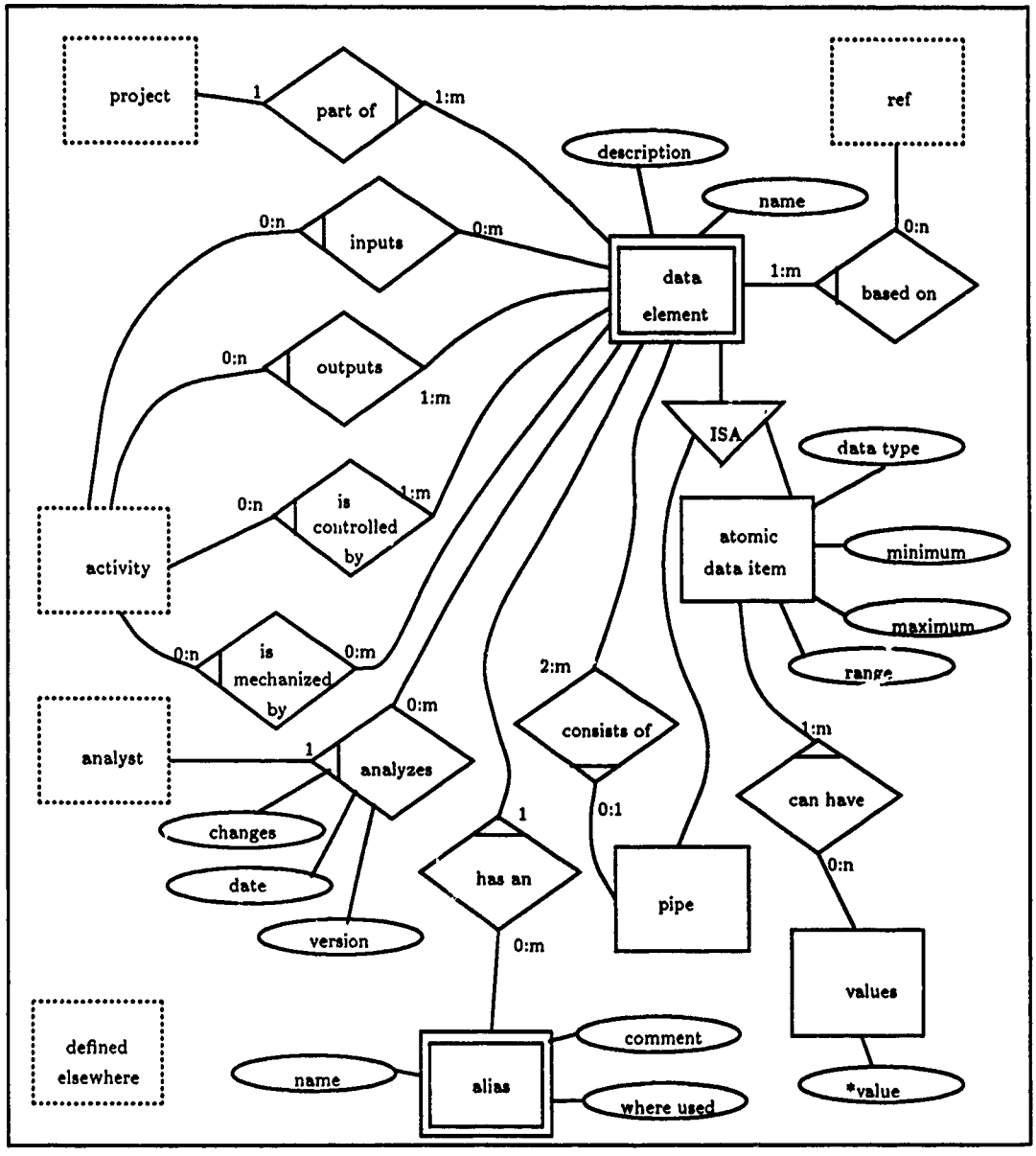
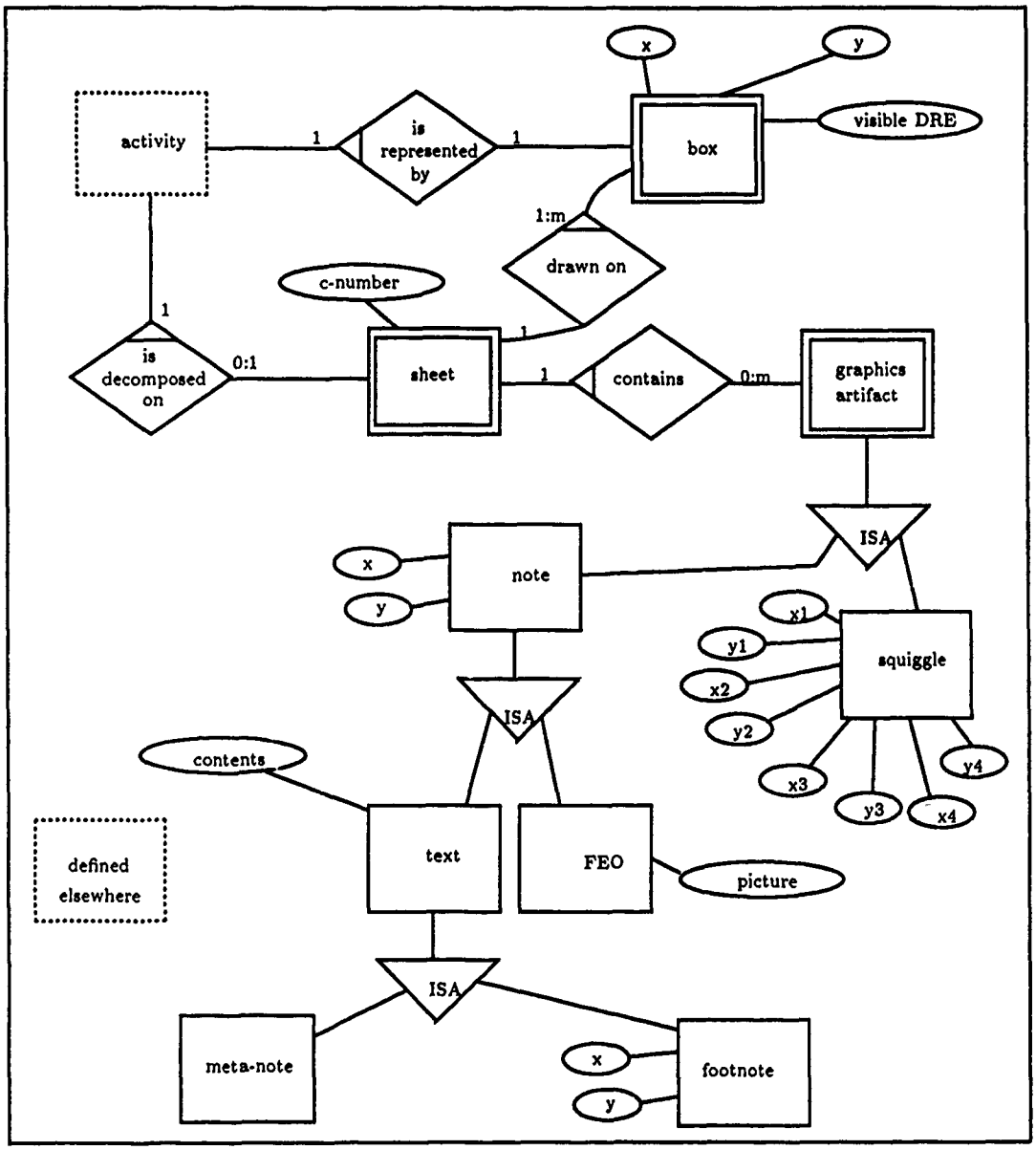


Figure 7. IDEF$_0$ Data Element Essential Data Model



(3:643)

Figure 8. IDEF$_0$ Activity Drawing Data Model

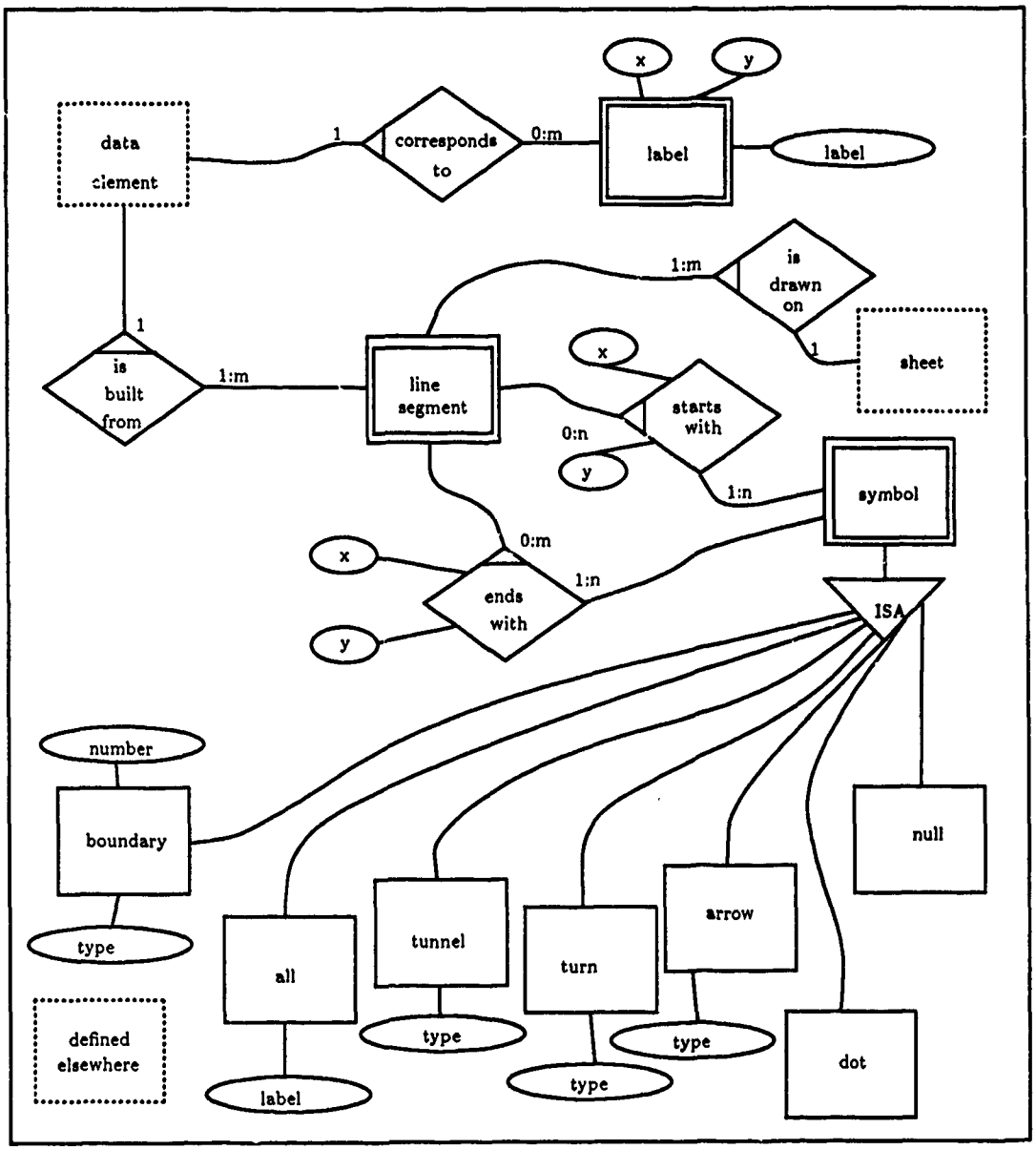


Figure 9. IDEF$_0$ Data Element Drawing Data Model

Mapping E-R Model Constructs to Objects in an OOD

One of the objectives of this thesis investigation is to perform a mapping from an E-R model to an OOD. OOD, at this time, is not a "complete" software life cycle methodology. In fact, it is mostly limited to the design and coding phases of the software life cycle (6:47). Thus, the software system analysis phase is commonly accomplished by employing one or more of the Yourdon Structured Method [1] (YSM) 2.0 [2] structured analysis graphical modeling techniques such as DFDs, E-R models, and State Transition Diagrams (STDs) (43). Other software system analysis modeling techniques for the systems analysis phase are utilized and defined in the literature but are invariably derivatives, relatives or combinations of the YSM 2.0 techniques. Once the software systems analysis is accomplished, various techniques are then employed to construct an OOD from these models. In this research, however, the mapping process from E-R models constructs to objects in an object oriented design is of interest and is examined more closely. In particular, the mapping of relationships to objects is emphasized.

The Keystone System Design Methodology. The Keystone System Design Methodology developed by Eric Kiem employs E-R models in performing a mapping to an OOD (23). The justification for using the E-R model follows:

> If E-R modeling of data for a database produces data structures with minimum duplication and minimum coupling, then an object oriented packaging schema resulting from the same model will exhibit the same characteristics – that is, a system developed using an E-R model will exhibit minimum duplication of data, and minimum inter-package coupling (23:102).

The methodology not only considers E-R model entities as objects, but also considers relationships between objects as objects themselves (23:102). Each entity and each relationship thus becomes an object in the OOD. The objects that model the relationships are the only objects which have

[1] Yourdon Structured Method is now a trademark of Yourdon, Inc (8).
[2] The various YSMs are now explicitly numbered (8).

visibility to (i.e., are coupled to) other objects in the final OOD. Their visibility is restricted to the objects which they relate. The objects which model the entities are not coupled and thus are considered as candidates for reuse (23:105).

Object Oriented Systems Analysis Method. The Object-Oriented Systems Analysis Method results in the creation of an *Information Model* of a system. The graphical representation of the Information Model is an *Information Structure Diagram* which is partly based on the E-R model developed by Chen (37:77). The Information Structure Diagram syntax refers to entities as objects and refers to relationships as either correlation tables or associative objects (37:79).

Correlation tables and associative objects are very similar. Correlation tables are modeled as relations that only contain the primary key fields of the objects (i.e., the entity relations) that they relate. Associative objects are the same as correlation tables except they contain one or more attribute fields in addition to the primary key fields.

The procedure for mapping objects and relationships in an Information Structure Diagram to objects in an OOD is not explicitly stated. However, the methodology requires *Object Specification Document* be produced which provides an object specification for "every object in the model"(37:83). The methodology requires that associative objects be included in the specification, whereas correlation tables are not included(37:83). Thus, it appears that correlation tables are not considered as objects.

An Earlier Mapping Methodology for SAtool II. Previous thesis work used the $IDEF_0$ Abstract Data Model as the basis for devising a mapping methodology from an E-R diagram to objects in an OOD (38). The following methodology is used to identify the objects for essential data model OOD:

1. Map all entities in an ER diagram into objects.
2. Combine a key object with a related object by defining the relationship between the two as an attribute of the main object. Thus, only one operation will be defined for the given relationship.
3. Map all relationships between two objects into two operations based on their two semantic interpretations. Then assign each operation to the appropriate object.
4. Map all attributes of an entity to the object that represents the entity.
5. Map all attributes of a relationship to the object that contains an operation that represents the relationship. (38:4–12 to 4–13)

Unfortunately, the implementation of this methodology leads to an OOD with only three highly coupled objects: a project, an activity, and a data element.

Expert Systems

As stated in Chapter One, the primary components of an expert system (also referred to as knowledge-based systems) are the inference engine and the knowledge base.

The paradigm of problem solving which underlies all expert systems and AI programs in general is search (9:160), and it is the purpose of the inference engine to employ one or more search methods (or reasoning methods) to the knowledge in order to solve a problem.

There are several different knowledge representation schemes for representing information in a knowledge base. These schemes fall into one of four categories (27:334):

1. Logical representation schemes. First order predicate calculus is the most common method.
2. Procedural representation schemes. The knowledge is represented as a set of instructions. For example, the *if....then* constructs of a rule-based system are typically used.
3. Network representation schemes. Examples include semantic networks, conceptual dependencies, and conceptual graphs.
4. Structured representation schemes. Examples include scripts, frames, and objects.

The knowledge representation scheme of interest for this research is the procedural scheme, since it is the scheme used by CLIPS/Ada (35:1).

The knowledge base of a rule-based expert system contains long-term static knowledge which is represented by rules and also facts that are static, true propositions (17:924). Separate from the knowledge base is a global or *working memory* which stores the case-specific data concerning the particular problem to be solved. It is also used to store the short-term intermediate results (i.e., temporary assertions) during the expert system execution cycle (17:924). The working memory can therefore be viewed as representing the state of the expert system problem solving process at any given time during the execution cycle.

To illustrate how rules and facts can be represented in a rule-based expert system, an example is provided using CLIPS/Ada. Suppose that the rule *'Any triangle is a polygon'* and the fact *'Figure 1 is a triangle'* are to be represented as a CLIPS/Ada rule and a CLIPS/Ada fact respectively. The fact that 'Figure 1 is a triangle' can be represented by a single line of text:

```
(triangle figure1)
```

The representation of the rule is slightly more complex. The first line simply gives the rule a unique name. Any lines on the "left hand side" of the rule (prior to the arrow) constitute the *if* part of the rule. Any lines on the "right hand side" of the rule (after the arrow) constitute the *then* part of the rule.

```
(defrule  polygon-checker
 (triangle ?x)
=>
 (assert  (polygon ?x)))
```

The inference engine of a rule-based expert system has three phases of execution which are commonly called the *recognize-act cycle* (27:131):

1. The *match* phase compares the left hand side of rules to facts in the working memory and determines which, if any, are eligible to fire (execute).

2. The *conflict resolution* phase decides which rule will fire next if more than one rule is eligible to fire.

3. The *act* or *fire* phase executes the right side of the rule selected to fire. This step may or may not affect the contents of working memory.

There are two common control methods to this cycle: backward chaining or forward chaining. Backward chaining systems are considered "goal driven", because they fire rules and assert facts only until one or more goals are satisfied. Forward chaining systems, on the other hand, are considered "data driven", because they typically fire rules and assert facts until all the eligible rules have been fired. For example, the inference engine of CLIPS employs a forward chaining reasoning method (35:128). If a CLIPS knowledge base contained the aforementioned "polygon" rule and the "triangle" fact, an execution of the CLIPS system would result in the addition of the following fact to the working memory:

```
(polygon figure1)
```

This result is accomplished by the left hand side of the "polygon" rule *matching* the "triangle" fact. Since there are no other rules eligible to fire, the inference engine directs the 'polygon-checker' rule to fire which creates the new fact in the working memory by executing the "assert" command contained in the right hand side of the rule.

Integration of Expert Systems with CASE Tools.

This section focuses on three examples related to the integration of CASE tools with expert systems: one from government academia (i.e., AFIT), one from civilian academia, and one from industry. The primary focus of the three projects reviewed will be in the area of expert assistance in the early phases of the software development life cycle (i.e., requirements analysis and design). By examining projects developed from three different perspectives, the strengths and weaknesses of each project can be identified for future reference.

Most of the early expert systems started as research projects built on stand alone machines (29:111). Today, however, expert systems are built on a variety of software and hardware platforms. Because of these various platforms, AFIT, academia, and industry have begun both theoretical and actual development of systems that integrate CASE tools with expert systems. Each of the following projects have been implemented with differing degrees of success.

SAtool with Syntax Validation. As previously discussed, Steven Johnson developed SAtool for his MS thesis in 1987 (19). SAtool, though, is simply a graphical editor and provides no advice or assistance to the user as the $IDEF_0$ diagrams are being drawn. In other words, the user of SAtool could not determine if the finished diagram is consistent with the $IDEF_0$ graphical language except by tedious and time consuming manual inspection.

In order to provide the user with expert assistance and hence increase the value of SAtool to the software engineer, AFIT sponsored a MS thesis by Jung in 1988 (20). This research focused on the prototype development of an $IDEF_0$ syntax (language) validation tool which is an expert system to perform a syntax validation of the an $IDEF_0$ diagram. The $IDEF_0$ syntax is formalized by converting the syntax to predicate logic facts. The research describes how both a box and an arrow are described in predicate logic.

> The graphical feature BOX is translated into the predicate BOX(x), which means: x is a BOX. In the case of the ARROW, it is translated into the predicate ARROW(x), which means: x is an ARROW (20:3-5).

There are two steps to the syntax validation tool (20:3-4). First, a C program is developed called a *translator* to translate the $IDEF_0$ diagram features into a formal description that is 'readable' by the expert system. The expert system is a backward chaining expert system, BC3[3]. BC3, however, required facts to be represented as three-element lists of the form [Object, Attribute, Value] which are normally referred to as OAV triples. The $IDEF_0$ diagram representation is stored

[3]BC3 is a Prolog based backward chaining expert system shell developed by F. M. Brown.

in multiple C data structures, and the translator program creates a file of facts based on the information in those structures (20:3–4 to 3–8).

The second and final step of the syntax validation tool is the *syntax checker* (20:3–4, 3–10). The syntax checker's purpose is to check the $IDEF_0$ diagram (now represented as OAV triples) for errors. The syntax checker is, in essence, the expert system. It consists of the fact base (provided by the translator program), a Prolog inference engine (BC3) to do the reasoning, and a set of $IDEF_0$ syntax rules. Syntax rules such as "Each box must have a name" and "Each arrow must have a label" are converted to *if....then* constructs in a form acceptable to BC3 (20:3–8). The syntax checker, when executed, produces error messages (if applicable) for the designer to review and take corrective action.

The research, however, is limited in scope. All the features of an $IDEF_0$ diagram are not addressed. Plus, a transparent integration of SAtool and the syntax validation tool is not achieved (i.e., a manual step remained).

The follow-on research of Kim centered on resolving both the aforementioned integration problem as well as expanding the syntactical checks that the expert system performed (24:1–2). The number of $IDEF_0$ syntactical features checked by the expert system are expanded (24:5-12 to 5-13). To resolve the integration problem, an attempt is made to integrate SAtool with a Quintus Prolog implementation of the syntax validator (the expert system). The "new" syntax validator is simply the expert system shell BC3 with changes necessary for it to run under Quintus Prolog. Unfortunately, compatibility problems between Quintus Prolog and the C programming language result in a failure to achieve a transparent integration of the expert system with SAtool (24:6–2).

Specification-Transformation Expert System (STES). At the University of Illinois, Tsai and Ridge have developed the Specification-Transformation Expert System (STES) which is an expert system that they have integrated with the CASE tool Teamwork developed by Cadre Technologies (41:34). Teamwork is used to create DFDs. In addition, Teamwork runs on an Apollo workstation

platform and includes a built-in Access tool which allows users to access the underlying data structures that contain the DFD description. In this case, a C++ program is written by Tsai and Ridge to access the DFD description (41:34).

By implementing the STES in OPS5. which can also run on Apollo workstations, transparent integration of Teamwork and STES is achieved.

After the requirements analysis phase of the software development life cycle is completed, STES can be used in the next step — the design phase. STES assists the software engineer with the design phase by transforming the DFD into a structure chart (41:28). The STES examines the C++ representation of the DFDs, extracts the salient features, and converts them into production rules (41:31). The STES then "applies inference to identify and transform the efferent, afferent, and transformation-centered components of the dataflow diagram into a first-cut structure chart" (41:31).

Visible Analyst Workbench. Visible Analyst Workbench [4] is an IBM-PC based CASE tool marketed by Visible Systems Corporation that contains *rules* to perform error checking of DFDs (42). According to the product documentation, the CASE tool portion called Visible Analyst allows the user the choice of two different styles in DFD construction: the Yourdon/DeMarco Method [5] DFD or the Gane and Sarson Method DFD (42:30,32). Unlimited levels of DFD process decomposition are also supported. Regardless of the style chosen, however, the rules portion of the tool called *Visible Rules* can check the diagram for proper balancing, naming conventions, etc (42).

The Visible Rules are executed without leaving the DFD which means transparent integration between the CASE tool portion and the "expert system" portion is achieved. Although the word *rules* implies a rule-based expert system is used, the proprietary nature of the product does not

[4] Visible Analyst is a registered trademark of Visible Systems Corporation.
[5] The correct reference should probably be YSM 1.0 (8).

permit the disclosure of whether the rules are implemented algorithmically or by an expert system paradigm.

Summary

This chapter provides a review of several subject matter areas that directly relate to this investigation. $IDEF_0$ and its AFIT modifications are reviewed, since the $IDEF_0$ modeling technique is implemented by SAtool II, which the subsystem created here is destined for integration with.

The abstract data model of the $IDEF_0$ language as well as the data dictionary formats based on the language are reviewed, because modified versions are used in this research as the primary system requirements documents for the design phase.

Expert systems are reviewed because of the desire for SAtool II to perform syntactical validation of $IDEF_0$ models. In addition, interest exists in the automatic generation of $IDEF_0$ models from only essential data model information. Also of interest is the issue of augmenting the knowledge base of the expert system with application specific design knowledge of the system being modeled.

Several examples concerning the integration of CASE tools with expert systems are also reviewed, since this research calls for a similar integration. Clearly, all attempts at integration do not succeed. To improve the chances of successful integration, information from successful projects should be obtained and used as a foundation for further research. The compatibility of the CASE tool and the expert system appears to be a key point to focus on when considering integration.

III. REQUIREMENTS ANALYSIS

Introduction

This chapter presents a review of the requirements for the subsystem that is to be integrated with SAtool II. Presented are a revised $IDEF_0$ Essential Data Model and a revised AFIT Data Dictionary format. These two models are used in this research as the primary system requirements for the design phase. An overview of the requirements specification and how this research fits into the overall development of SAtool II are presented first. Next, the inconsistencies and the inadequacies of both the $IDEF_0$ Essential Data Model and the AFIT Data Dictionary format are identified. The revisions to the $IDEF_0$ Essential Data Model are then discussed, and a revised model is presented. The revisions to the AFIT Data Dictionary formats are detailed and are followed by revised formats. The revised $IDEF_0$ Drawing Data Model from (40) is also included to present a total picture of the revised $IDEF_0$ Abstract Data Model. Finally, a discussion of the expert system requirements is presented including justification for the selection of CLIPS/Ada as the expert system tool.

Overview

The overall goal of both this research and (40) is to develop an Ada based CASE tool (SAtool II) for the creation, editing, output, storing, and syntax checking of $IDEF_0$ diagrams. Features for editing and storing AFIT Data Dictionary entries are also desired. The resulting CASE tool would then demonstrate the feasibility of Ada in a CASE tool environment.

In addition to formalizing the $IDEF_0$ syntax language, another stated purpose of the $IDEF_0$ Abstract Data Model presented in Chapter Two is to provide a model for "the implementation of an object-based Ada tool using this model" (3:643). Therefore, the design and implementation of SAtool II is heavily based on that model. However, the abstract data model only models the

$IDEF_0$ syntax, and there are many other system requirements. Therefore, the following is a general overview of the SAtool II system requirements specification:

1. The tool must be totally implemented in Ada.
2. The tool must have a graphical user interface (GUI).
3. The tool must be implemented on a workstation supporting X-Windows and Ada.
4. The tool must provide for the creation, editing, and output of $IDEF_0$ diagrams (i.e., the manipulation of $IDEF_0$ syntax).
5. The tool must provide a for the creation, editing, and output of the AFIT Data Dictionary formats.
6. The tool must provide for the storing of the essential data model information of an $IDEF_0$ model that is separate from the stored drawing data model information.
7. The tool must provide for the storing or automatic generation of the drawing data model information (i.e., the diagrams) of an $IDEF_0$ model that is separate from the stored essential data model information.
8. The tool must be integrated with an Ada based expert system for the purpose of identifying $IDEF_0$ syntax and modeling errors.
9. The tool must allow for a user to terminate work on an $IDEF_0$ model, leaving it in an unfinished state. For example, creating an activity with no connecting data elements leaves the $IDEF_0$ model in an incomplete state.
10. The tool must be developed using an object oriented design methodology in order to assess its potential in the construction of an Ada based CASE tool.

Both this investigation and (40) must satisfy requirements 1, 4, 9, and 10. This investigation also focuses on those requirements relative to the essential data model, the data dictionaries, and

the expert system (i.e., 5, 6, and 8). The concurrent research (40) focuses on those remaining requirements related to the drawing data model, the $IDEF_0$ diagrams, and the GUI (i.e., 2, 3, and 7). No specific requirements in terms of time or space are provided. It is recognized, however, that the response time of a CASE tool is critical to its acceptance by the user community. Therefore, efficiency in terms of time complexity is a factor in the design and implementation process.

A review of the requirements suggests that there are actually four subproblems to be solved by this research.

1. The development and implementation of an object model to create, retrieve, and restore $IDEF_0$ Essential Data Model information (based on requirements 1, 4, 6, 9, and 10).
2. The development and implementation of a method to create, retrieve and output AFIT Data Dictionary information (based on requirement 5).
3. The development and implementation of an interface to an Ada based expert system (based on requirement 8).
4. The integration of the above into a single subsystem for eventual placement within SAtool II (the primary requirement).

Henceforth, the term *Essential Subsystem* is used to refer to the single integrated subsystem designed for solving these four subproblems.

Alternative design and implementation methods using relational and nested-relational databases for representing the essential data model data and drawing data model information are explored in (31) and further research is planned by AFIT in this area. The speed at which the information can be stored and retrieved using this methodology is the primary barrier to its adoption.

A Review of the Requirements Models

Since the requirements for the solutions to the first and second subproblems are illustrated by the $IDEF_0$ Essential Data Model and the AFIT Data Dictionary formats respectively, a review of these models is performed prior to beginning the object oriented design process.

Prior to the start of this research, both the $IDEF_0$ Essential Data Model and the data dictionary formats were considered to be "complete". However, in the course of reviewing the essential model and the data dictionary formats, some inconsistencies between the two models and inadequacies in the models are noted. Therefore, before beginning the OOD phase, revisions are made to both models, and the new versions are presented.

The term *inconsistent* used in this context means contradictory. For example, a data dictionary normally provides a link between a graphical representation of a system and a descriptive representation of a system. Therefore, entities present in a graphical representation of a system like an E-R model should have corresponding descriptive information in a data dictionary. Thus, an entity in the graphical model without an entry in the data dictionary is inconsistent with the modeling technique. Likewise, an entry in the data dictionary without a corresponding entity in the E-R model is also considered inconsistent.

The term *inadequate* in this context means that some element of the model is present but is lacking in necessary information. For example, a data dictionary entry should, at a minimum, convey enough descriptive information about an entity or object of a system so that the entity or object is accurately represented by the description. In fact, the data dictionary should convey additional information that cannot be derived from its graphical counterpart. Thus, if the descriptive information is inadequate, then the data dictionary entry is considered inadequate as well.

Inconsistencies in the IDEF$_0$ Essential Data Model and Data Dictionary Format

By visual inspection of the essential model and the data dictionaries, several inconsistencies are identified. Figures 3 and 4 clearly show the presence of an *alias* field for both the activity and the data element data dictionary entries. Figure 7 contains an *alias* entity corresponding to its data dictionary. However, an examination of Figure 6 reveals no corresponding entity for its *alias* attribute in the data dictionary. This results in an inconsistency between the two models depicted in Figures 3 and 6.

The second inconsistency is also discovered by visual inspection of the E-R models and the data dictionaries. The IDEF$_0$ Activity Essential Data Model in Figure 6 clearly shows the presence of a *historical activity* entity. Its presence is consistent with the IDEF$_0$ language which permits activities to have a downward pointing mechanism arrow (known as a 'call') that refers to one or more activities in the same or other projects (30:3–10). A visual examination of the activity data dictionary in Figure 3, however, reveals no entry for historical activities, and thus, another inconsistency is identified.

Not only are inconsistencies identified between the IDEF$_0$ Activity Essential Data Model and the activity data dictionary entry, but a greater inconsistency is discovered between the IDEF$_0$ language and both the IDEF$_0$ Abstract Data Model and the AFIT Data Dictionary. Again, the inconsistency pertains to the use of aliases. Neither the IDEF$_0$ language nor the IDEF$_0$ modeling technique discussed in (30) make any reference to the use of aliases. Thus, an attempt to model a nonexistent feature of the IDEF$_0$ language is made.

Inadequacies in the IDEF$_0$ Essential Data Model and Data Dictionary Format

The inadequacies present in both the essential model and the data dictionary are difficult to identify. In particular, inadequacies in the essential model are difficult to determine due to the fact that it is an abstract graphical model of the IDEF$_0$ language. One cannot simply "plug"

the syntactic values of any specific $IDEF_0$ model into a graphical E-R model because of its level of abstraction. Therefore, the methodology developed for determining inadequacies involved modeling $IDEF_0$ diagrams with data dictionaries. The following steps illustrate that methodology:

1. Create a set of $IDEF_0$ diagrams that model a system.
2. For each activity and data element contained in the diagrams of the model, create a data dictionary entry and fill in the corresponding values from the diagrams.
3. Engage a second person with no knowledge of the original $IDEF_0$ model to try and recreate the original $IDEF_0$ model using only the data dictionaries.
4. A successful recreation of one of the equivalent drawing models means a failure in the search for inadequacies; a failed recreation attempt means one or more inadequacies are present in the data dictionary.

The last step in the methodology highlights a crucial theoretical aspect of the essential model and its accompanying data dictionary. If the data dictionary correctly describes the entities and relationships of the essential model, then for a given data dictionary that models an actual $IDEF_0$ model, a human should be able to recreate one of the several equivalent $IDEF_0$ models by only examining the data dictionary. In other words, the human should be able to create one of the many drawing models that represent the same underlying essential model. If the human is unsuccessful in creating one diagram from an equivalent set of $IDEF_0$ diagrams, then inadequacy(s) must exist in the data dictionary and/or the essential data model. The issue of whether a computer can successfully recreate one or more of the drawing models from only essential model information is a complexity issue that is not addressed in this research.

Using the methodology described above, two significant inadequacies are identified:

- The inability to correctly model multiple Source-Destination groupings.
- The inability to correctly model multiple decompositions of a data element.

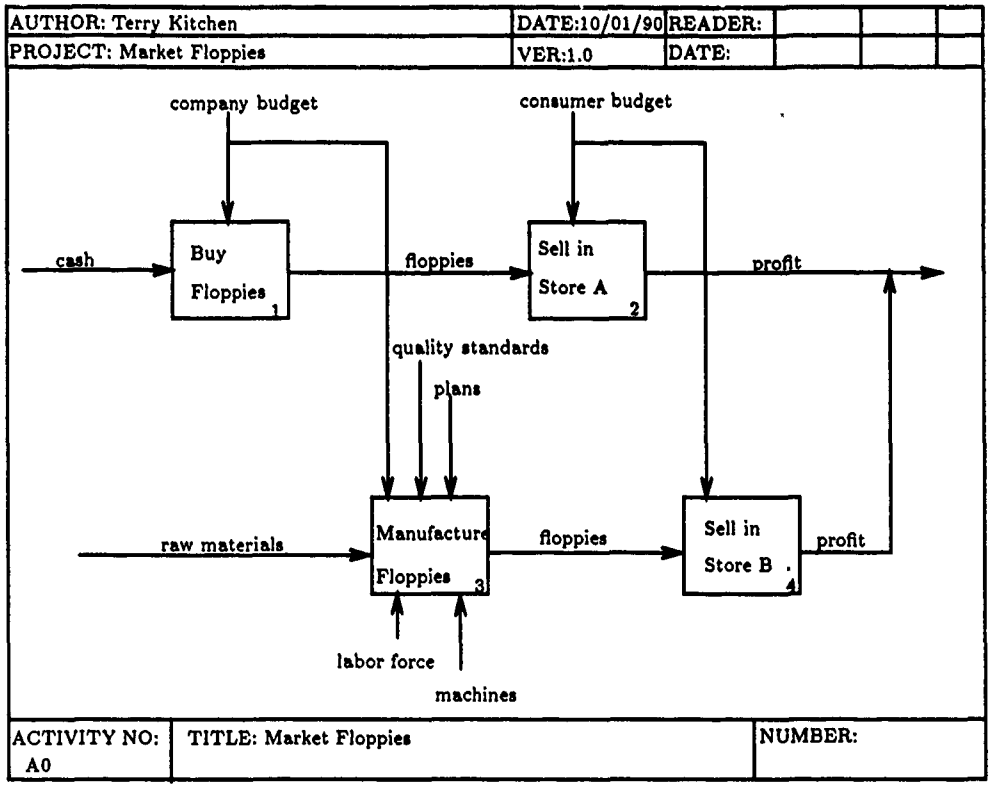


Figure 10. $IDEF_0$ Diagram Illustrating Multiple Source-Destination Groupings

Multiple Source-Destination Groupings. Multiple Source-Destination groupings only occur when the same data element appears in two different relationships within the same $IDEF_0$ diagram. Figure 10 shows that the data element 'floppies' appears in two separate relationships within the diagram 'Market Floppies'. In this example, 'floppies' actually has *multiple source-destination pairs.*

Figure 11 is an abbreviated data element data dictionary entry for 'floppies' which illustrates the inadequacy of the current data element data dictionary format in correctly modeling Figure 10.

In this case, the data dictionary entries for each of the four activities and even the parent diagram provide no additional information as to whether floppies that are bought are resold to store A or B. Likewise, are manufactured floppies sold in store A or B? The answer cannot be derived

NAME:	Floppies
TYPE:	Data Element
SOURCES:	Buy Floppies
	Manufacture Floppies
DESTINATIONS:	
INPUT:	Sell in Store A
	Sell in Store B
CONTROL:	

Figure 11. Abbreviated Data Dictionary Entry for Data Element 'Floppies'

NAME:	Floppies
TYPE:	Data Element
SOURCES:	Buy Floppies
DESTINATIONS:	
INPUT:	Sell in Store A
CONTROL:	
SOURCES:	Manufacture Floppies
DESTINATIONS:	
INPUT:	Sell in Store B
CONTROL:	

Figure 12. Revised Abbreviated Data Dictionary Entry for Data Element 'Floppies'

from data dictionary and/or the essential model. Note, however, that if an interpretation of the $IDEF_0$ language disallows the same data element appearing more than once on the same diagram, no changes to either of the models is required.

To correct the situation, the data dictionary must be able to distinguish between different groupings or pairs of sources and destinations. Figure 12 illustrates a solution which mandates a separate list of Sources and Destinations for each appearance of a data element. For instance, if the data element 'floppies' also appeared once on another diagram associated with Figure 10, then a total of three Source-Destination groupings would appear in Figure 12. With the revised format, there is no doubt what the correct sources and destinations for 'floppies' are.

Multiple Decompositions of Data Elements. Multiple decompositions of a data element occur when a given data element is decomposed into one set of subcomponents and then decomposed

into at least one other, but different, set of subcomponents within the same $IDEF_0$ model. In other words, unlike activities which can have only a single parent, a data element can have one or more parents. This feature of the $IDEF_0$ language is overlooked by Morris in his description of the 'consists of' relationship in the essential data model of Figure 7.

> This relationship shows that a pipe consists of at least two data elements and that a data element can be contained within at most one pipe (31:43).

Since Morris played a key role in the development of the $IDEF_0$ Abstract Data Model, and the E-R models presented in (31) are identical to the ones presented in Chapter Two, it appears that the intent of the E-R model is not to handle multiple decompositions of data elements. In fact, an inadequacy in the $IDEF_0$ Abstract Data Model does exist, since an additional attribute is required to track multiple decompositions.

The data element data dictionary format suffers from the same inadequacy as the data element essential data model. Figure 13 shows an abbreviated version of the current data element data dictionary format. The 'PART OF' field indicates what data element (if any) this data element is a part of. Note that it is classified as a single field (i.e., only a single entry is permitted). Obviously, if a data element is to have multiple parents, this data dictionary entry is inadequate in modeling that feature of the $IDEF_0$ language. Like the previous inadequacy, a "grouping" of fields in the data dictionary entry is necessary to correct the inadequacy illustrated in Figure 13. The format for this new grouping appears in the revised data element data dictionary format illustrated in Figure 21.

Junctors. Assuming that the previous two inadequacies are corrected, a possible third inadequacy pertaining to modeling of junctors is considered but not resolved. Junctors are the points where two or more arrows with different labels meet. It is argued that in order for an $IDEF_0$ model to be recreated from the essential data model information, the junctors must be modeled

(S) NAME:	(data element name)	C25
(S) TYPE:	(defaults to DATA ELEMENT)	N/A
(S) PART OF:	(parent data element name)	C25
(M) COMPOSITION:	(subcomponent data element names)	C25

Figure 13. Abbreviated Data Dictionary Entry Format for Data Element

somewhere within the $IDEF_0$ Essential Data Model and data dictionary. Several test cases were developed, but a human test subject was always able to successfully recreate one of the equivalent $IDEF_0$ diagrams. It is hypothesized that a data dictionary with the previous problems corrected can be developed such that the original $IDEF_0$ model can not be successfully derived from the data dictionary. However, it is not one of the objectives of this research to prove or disprove the hypothesis. Thus, the issue remains open for future research.

A Revised $IDEF_0$ Abstract Data Model

In this section, changes to the essential model are discussed and revised E-R models are presented for both the essential and the drawing models. A table is also included to describe the E-R constructs. The drawing models are from (40) and are included here in order to present a "complete" picture of the revised $IDEF_0$ Abstract Data Model.

Figure 14 illustrates the revised $IDEF_0$ Activity Essential Data Model. The following changes are made to the model:

1. The entity 'analyst' and the relationship 'analyzes' are dropped since the problem domain is restricted to only $IDEF_0$ language constructs. The attributes for both 'analyst' and 'analyzes' are then modeled as attributes of the activity entity.
2. The 'name' of an activity is made the key attribute of the activity entity, since the previous model lacked a key.
3. The attribute 'node number' is renamed to 'activity number' throughout the model.

Figure 15 illustrates the revised IDEF$_0$ Data Element Essential Data Model. The following changes are made to the model:

1. The 'alias' entity and the relationship 'has an' are dropped, because aliases are not consistent with the IDEF$_0$ language.
2. The entity 'analyst' and the relationship 'analyzes' are eliminated in the same manner as the activity essential data model.
3. To permit multiple decompositions of data elements, the entities 'pipe' (or composite data item) and 'atomic data item' are dropped, and the relationship 'consists of' is modified to reflect how data elements relate to one another. The fact that a data element can have multiple parents is reflected by the 'm to m' relationship. This differs from the "composed of" relationship which has a '1 to m' relationship indicating only one parent. The multiple decompositions are then tracked via the addition of the attribute 'decomposition id'.
4. The 'name' of a data element is made its key attribute.

Table 1 provides a description for each of the activity and data element essential data model constructs. Figures 16, 17, and 18 depict the revised IDEF$_0$ Drawing Data Model (40). Figure 19 illustrates the use of some of the entities and their relationships (40).

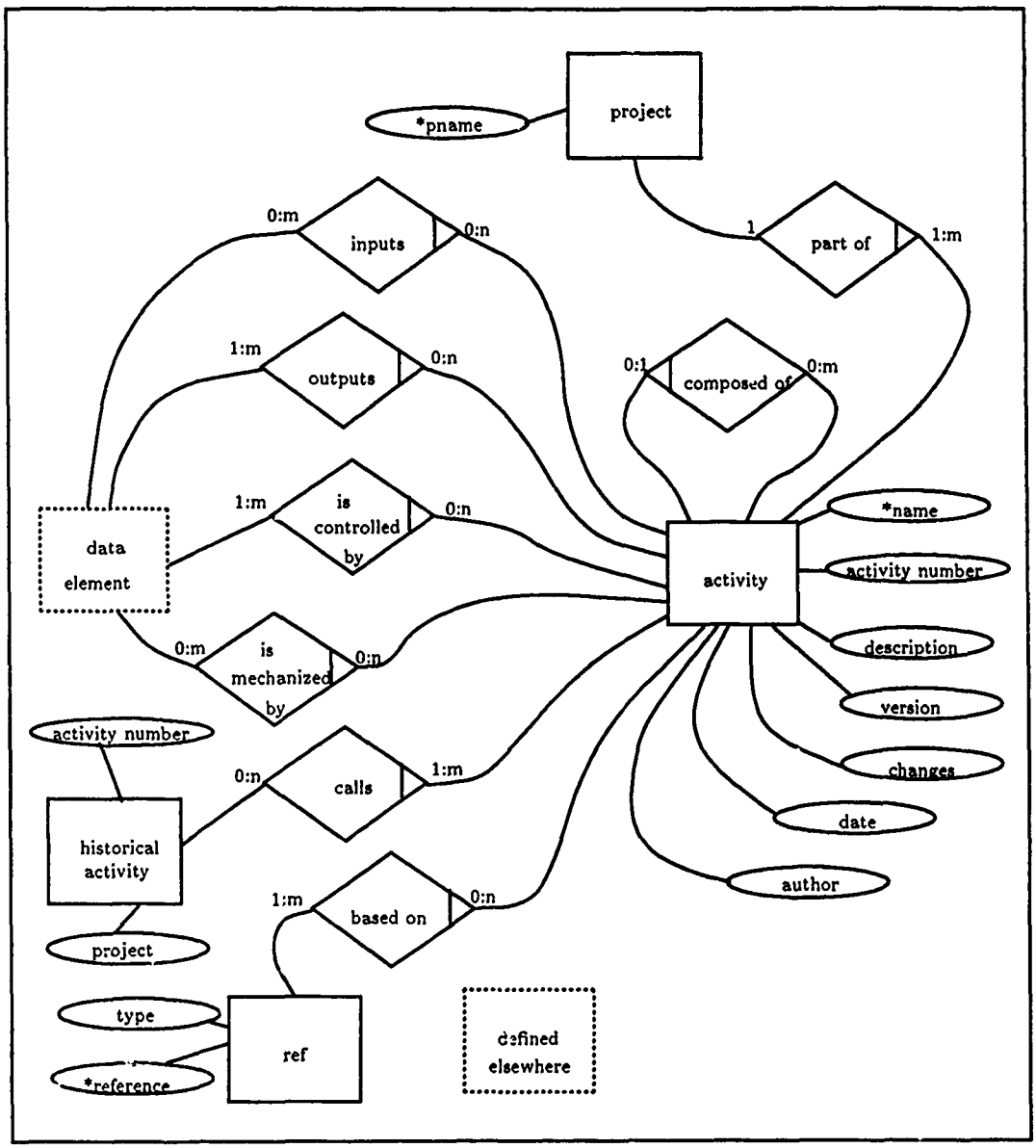


Figure 14. Revised IDEF$_0$ Activity Essential Data Model

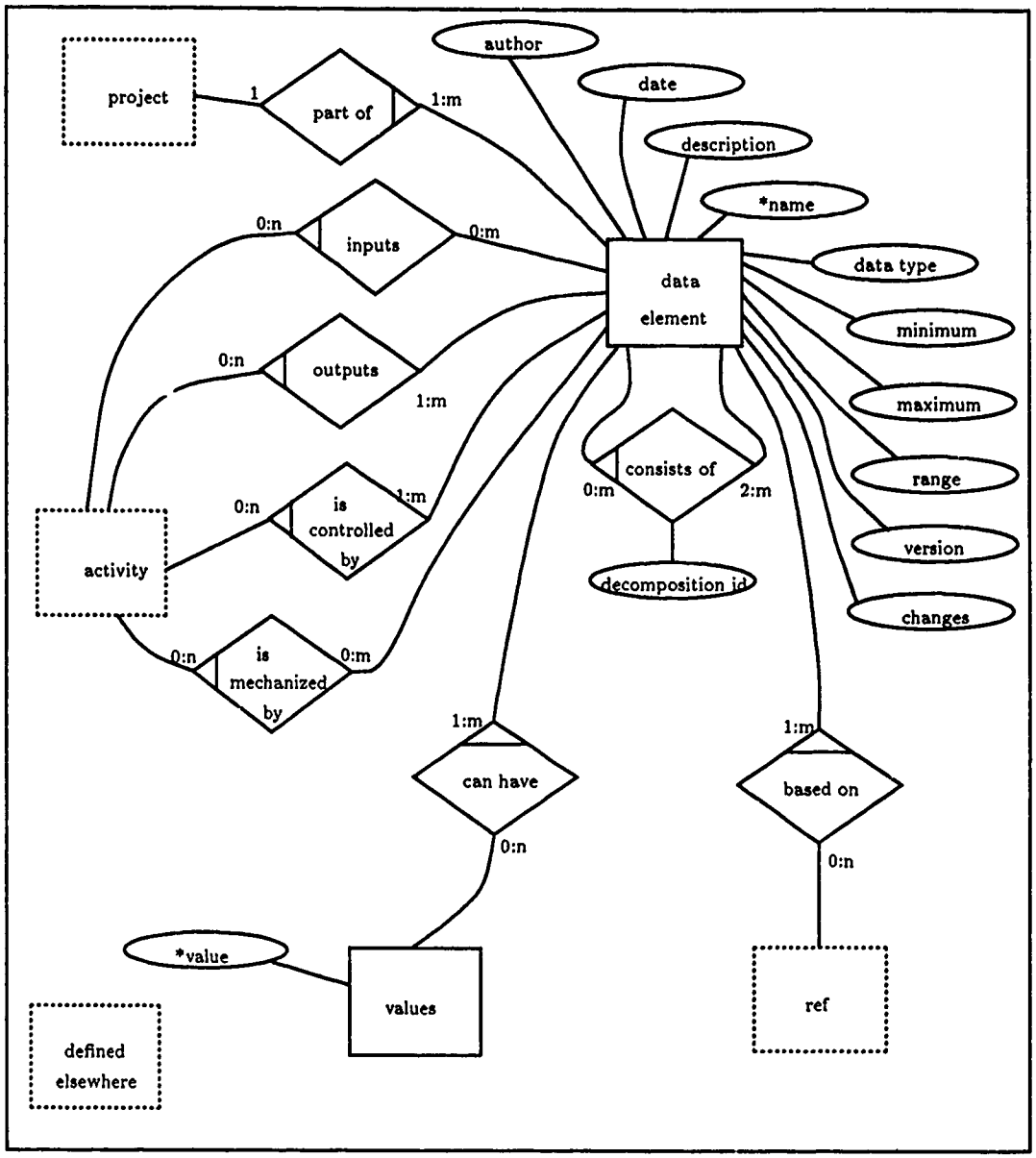


Figure 15. Revised IDEF$_0$ Data Element Essential Data Model

Table 1. Description of Components in the Essential Data Model

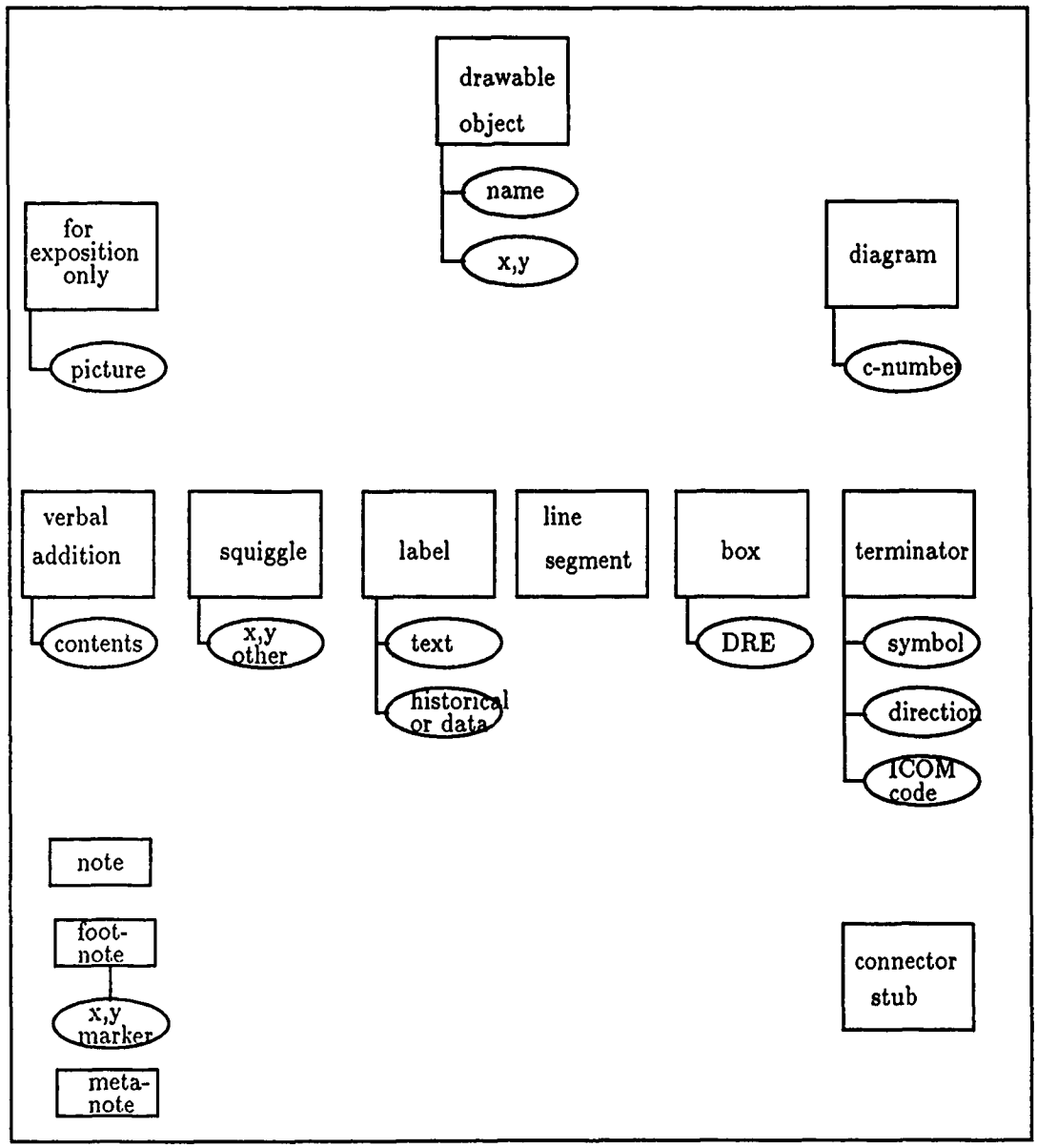
E–R construct	Description
activity	This entity, which is existence dependent upon **project**, represents the $IDEF_0$ activities. Attribute *name* is the name of the activity and its primary key, and *activity number* captures the dominance of one activity over another. Attribute, *description*, allows the analyst to describe the activity. Attribute, *version*, is used to record the current version number of the activity; *date* indicates when the activity is created; *changes* captures the change information about a given activity; *author* captures the author name of this activity (16:12).
composed of	This relationship shows that a given parent activity is composed of zero to many (0:m) child activities. It also shows that each activity has one parent activity. The 0:1 notation accounts for the fact that the A-0 activity does not have a parent activity (16:12).
project	This entity identifies the project to which each activity or data element is assigned. Key attribute, *pname*, indicates the name of the project (16:12).
part of	This relationship indicates that an activity or data element is part of exactly one project, whereas a project contains one to many activities or data elements.
ref	This entity captures any references associated with an activity or data element. Key attribute, *reference*, identifies which reference is involved, and attribute, *type*, identifies the type of reference (16:12). This entity allows a library of various documents such as DOD standards, user requirements, contractual clauses, etc., to be tied to the given activity or data element.
based on	This relationship indicates that a given activity or data element is based on one to many references, and that a given reference is the basis for zero to many activities or data elements.
historical activity	This entity is used to represent an activity in another project that is "called" by an activity in this project. Attribute, *project*, indicates which project contains the historical activity, and attribute, *activity number*, identifies the specific activity within the project.
calls	This relationship indicates the fact that an activity can call zero to many previously completed (historical) activities, and that a given historical activity is called by one to many activities (30:3– 11).

based on (31:42–43)

Table 1 (continued): Description of Components in the Essential Data Model

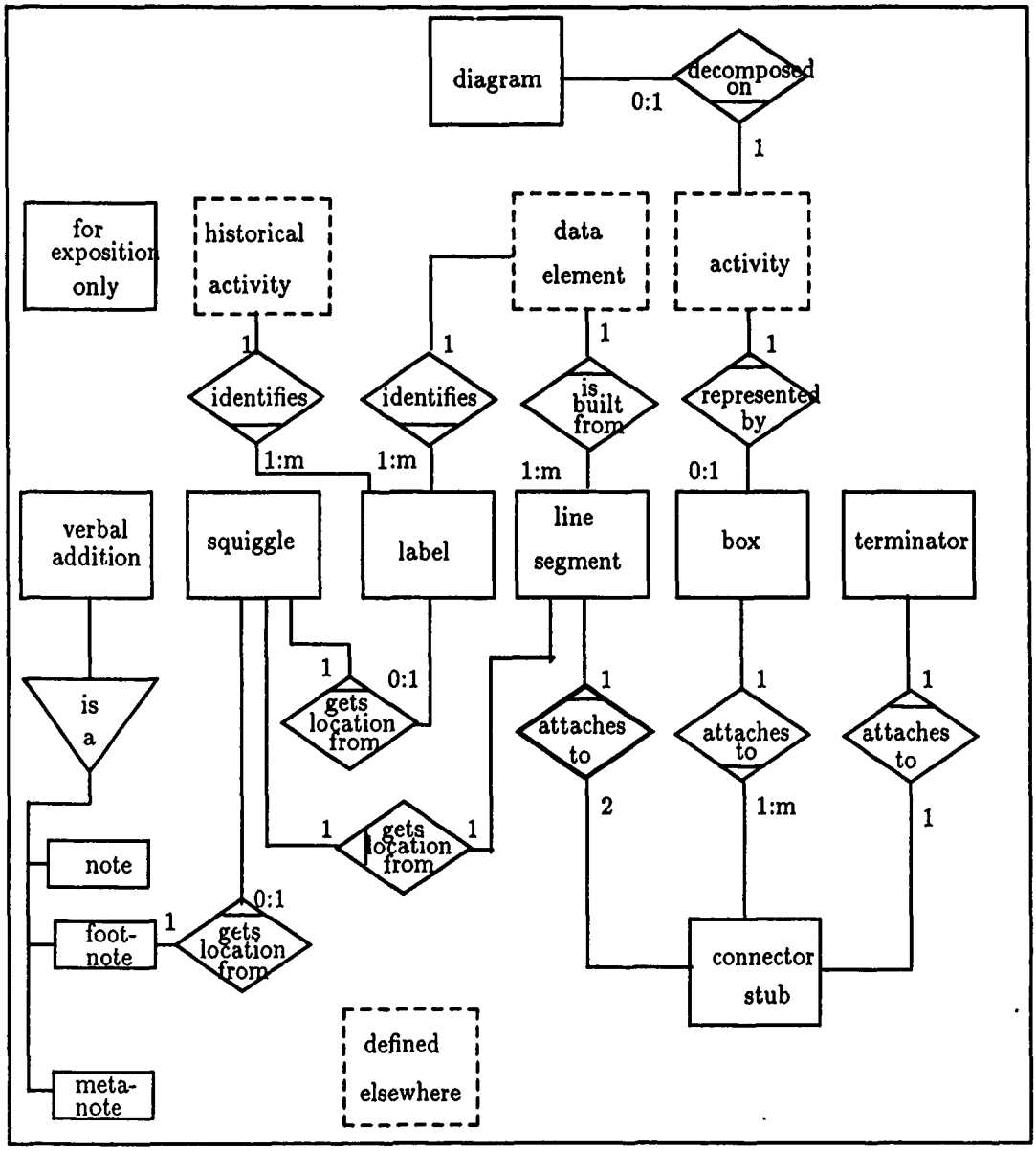
E–R construct	Description
inputs	This relationship indicates that an activity can input zero to many data elements. $IDEF_0$ only requires activities to have control data elements and output data elements (30:6–26).
outputs	This relationship shows that an activity must have at least one but can have many output data elements (30:6–26).
is controlled by	This relationship shows that an activity can have one to many control data elements (30:3–4).
is mechanized by	This relationship indicates that an activity can have zero to many mechanism data elements. $IDEF_0$ only requires activities to have control data elements and output data elements (30:6–26).
data element	This entity, which is existence dependent upon **project**, represents the $IDEF_0$ data elements. Key attribute, *name*, is the name of the data element. Attribute, *data type*, indicates the type of data (in the Pascal or Ada sense); *minimum* is the minimum data value, if applicable, *maximum* is the maximum data value, if applicable, and *range* is the data value range, if applicable (16:14). In the case that the data element consists of other data elements, none of the aforementioned attributes except *name* apply. In the case that the data element does not consist of other data elements and none of the attributes still are applicable, entity **values**, as described below, probably applies. The remaining attributes are applicable in any case and are identical to those of activity except that they, of course, relate to a data element.
consists of	This relationship shows that a data element can consist of two or more data elements, and that a data element can be contained within multiple data elements.
values	This entity is used for data elements which do not consist of other data elements and which have enumerated values, e.g., color can have values red, blue, and green. The entity has a single (key) attribute, *value* (16:14).
can have	This relationship ties a data element with no subcomponents to its corresponding values entity.

based on (31:42–43)



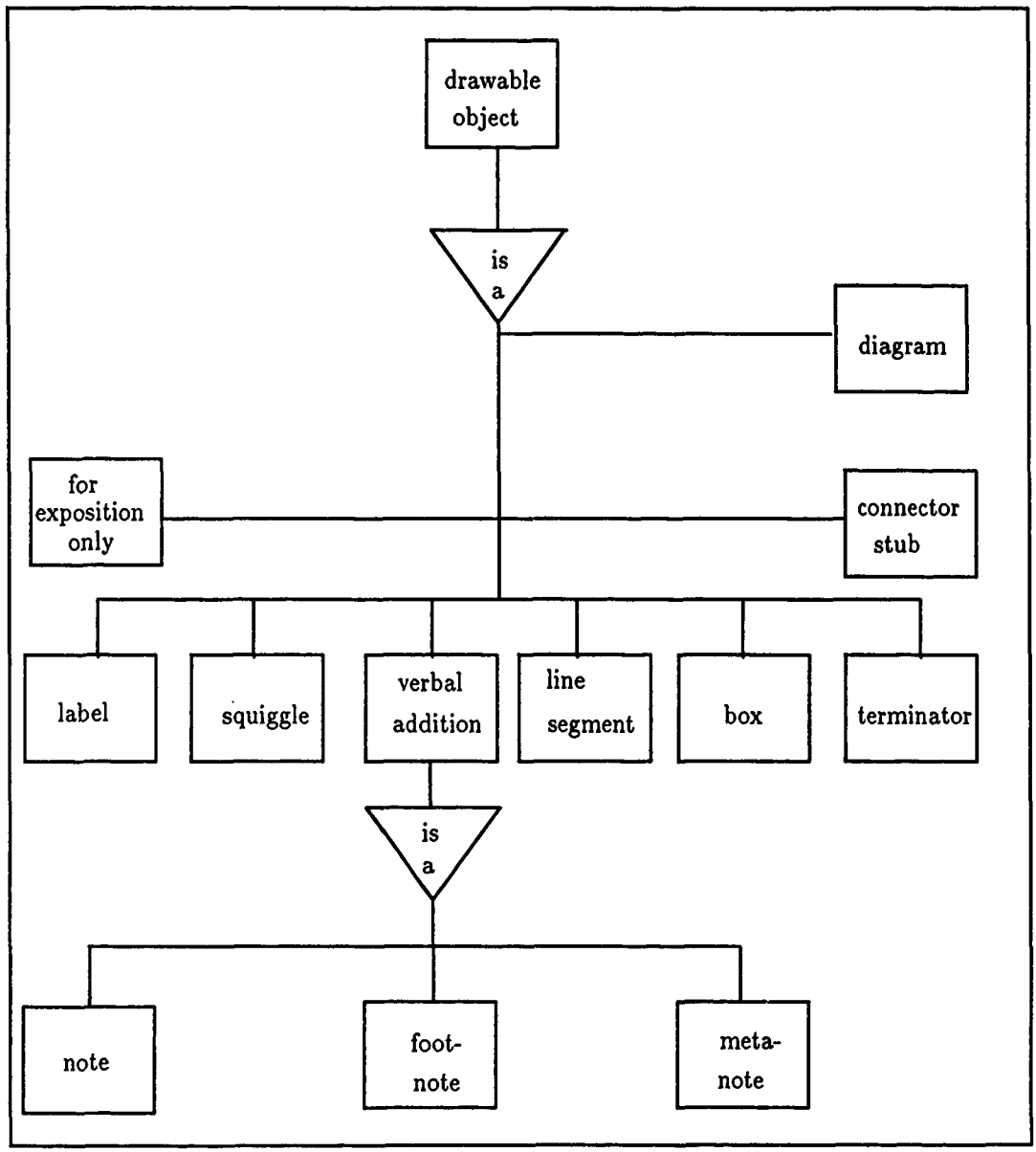
(40)

Figure 16. Revised IDEF$_0$ Drawing Model (Entities and Attributes)



(40)

Figure 17. Revised IDEF$_0$ Drawing Model (Entities and Relationships)



(40)

Figure 18. Revised IDEF$_0$ Drawing Model (Classes)

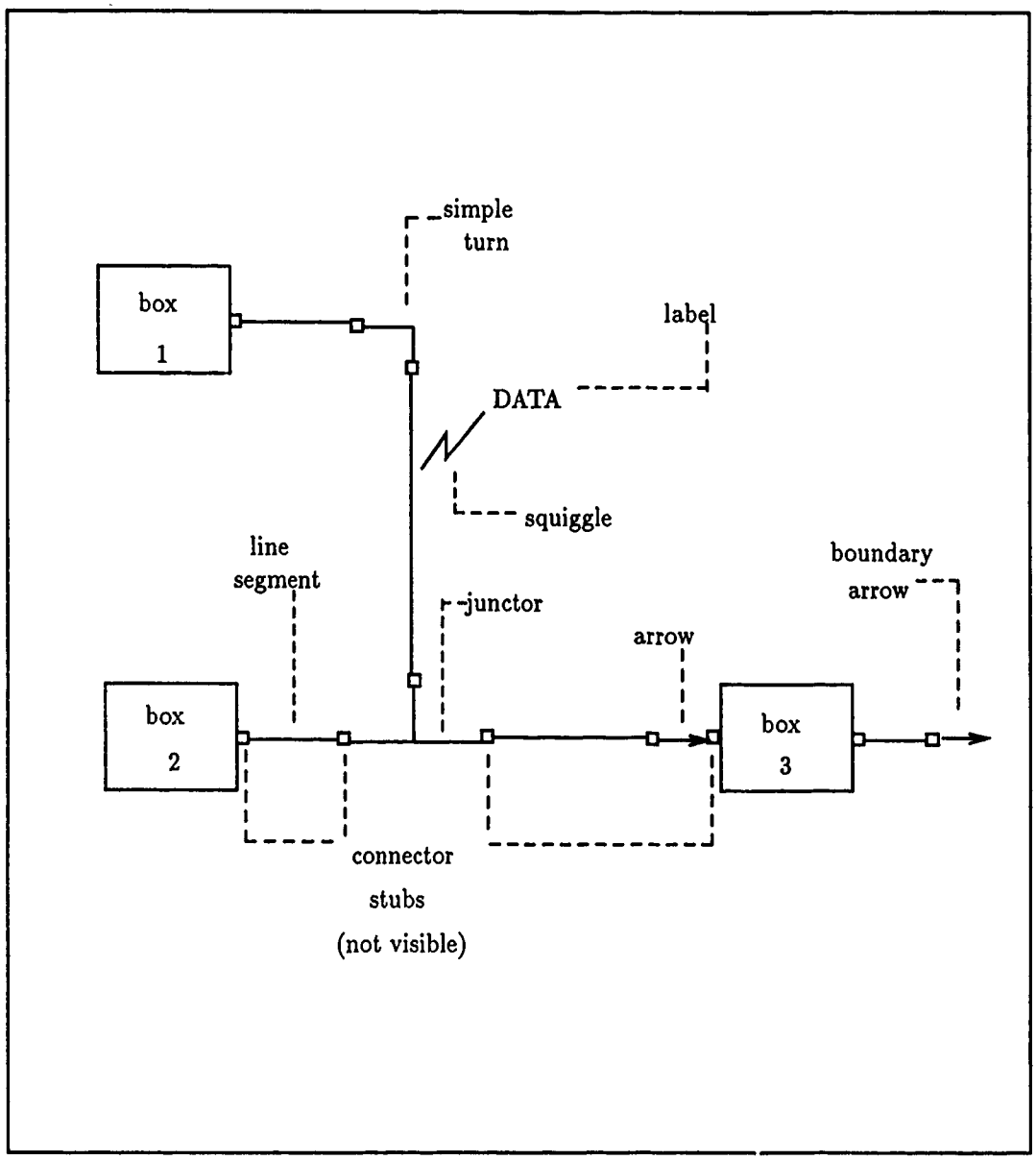


Figure 19. IDEF$_0$ Drawing Model Illustration

A Revised AFIT Data Dictionary Format

In this section, changes to the AFIT Data Dictionary formats are discussed and revised models are presented for both the activity and the data element models. In order to make the necessary modifications to correct the inconsistencies and inadequacies found in the data dictionary formats, an additional field classifications are necessary to precede some of the fields.

As discussed in Chapter Two, the 'S', 'M' or 'G' that precedes the fields in the data dictionary formats refer to terms 'single line field', 'multiple line field', and 'group field' respectively. An extension to these three field classifications called 'MG' is discussed in (11). The classification 'MG' means a "multi-line field within a group- field"(11:33). However, the research includes no syntactical definition for the classification. Therefore, the total revised field classifications for the AFIT Data Dictionary Format are as follows:

- (S) means that the field consists of a single field that appears on a single line.
- (ML) means that the field consists of a single field that can appear on one or more lines.
- (MF) means that the field consists of one or more fields, and each field is a single field that appears on a single line.
- (G) means that the field consists of two or more fields grouped together and multiple groups are allowed. However, each group member is still a single field that can only appear on a single line.
- (MG) means two or more fields are grouped together and multiple groups are allowed. Each group member is permitted to be a single field, a single field of multiple lines, or multiple fields. Therefore, each group member must be classified with either a 'S', 'ML', or 'MF' field classification.

Note that the two field classifications 'MF' and 'ML' resolve the ambiguity problem with the 'M' classification that was discussed in Chapter Two by distinguishing between a single entry with multiple lines (ML) and an entry that can consist of multiple fields (MF).

Figure 20 illustrates the revised data dictionary format for an activity. The following is a list of changes that are made to the data dictionary entry for activities presented in Chapter Two:

1. The ALIASES and COMMENTS fields are dropped due to the aforementioned inconsistency with the $IDEF_0$ language.
2. The group-field CALLS is added to account for the "calls" that an activity can make to one or more historical activities. This corrects an inconsistency with the data dictionary and the activity essential data model.
3. The VERSION CHANGES field is renamed to just CHANGES and reclassified from a single field only to a single field with multiple lines. The intent of this change is to allow for additional information about the version to be stored. However, the entry can now be used to store the history of the version changes.
4. The character length of the PROJECT field is extended to 25. This change is made because a project is, in essence, a parent activity, and the name lengths should be compatible. The original limit of only 12 characters is also considered to be slightly restrictive.
5. The character length of the AUTHOR field is extended to 25 characters to allow for greater flexibility.

Figure 21 illustrates the revised data dictionary format for a data element. The following is a list of changes that are made to the data dictionary entry for data elements presented in Chapter Two:

(S) NAME:	(activity name)	C25
(S) TYPE:	(defaults to ACTIVITY)	N/A
(S) PROJECT:	(project name)	C25
(S) NUMBER:	(activity number of this activity)	C20
(ML) DESCRIPTION:	(text description)	C60
(MF) INPUTS:	(data element name)	C25
(MF) OUTPUTS:	(data element name)	C25
(MF) CONTROLS:	(data element name)	C25
(MF) MECHANISMS:	(data element name)	C25
(G) CALLS:		N/A
PROJECT:	(different or same project name)	C25
ACTIVITY NUMBER:	(activity number in the project)	C25
(S) PARENT ACTIVITY:	(activity name of parent)	C25
(ML) REFERENCE:	(cite a reference to the activity)	C60
(S) REF TYPE:	(the type of the reference)	C25
(S) VERSION:	(version of this entry)	C10
(ML) CHANGES:	(a history of the changes)	C60
(S) DATE:	(mm/dd/yy of this entry)	C8
(S) AUTHOR:	(author's name)	C25

Figure 20. Revised Data Dictionary Entry Format for Activity

1. The ALIASES, WHERE USED, and COMMENT group field is dropped due to the aforementioned inconsistency with the $IDEF_0$ language.

2. The fields PART OF and COMPOSITION are added to a new multi-line group-field called DECOMPOSITIONS. This modification corrects the inadequacy in modeling multiple data element decompositions.

3. The fields SOURCES, INPUTS, and OUTPUTS are added to a new multi-line group-field called SOURCES/DESTINATIONS. This modification corrects the inadequacy in modeling multiple source destination groupings of the same data element on the same diagram.

4. The VERSION CHANGES field is changed in the same manner as the activity data dictionary entry.

5. The PROJECT field is lengthened to 25 characters for compatibility with the activity data dictionary entry.

(S) NAME:	(data element name)	C25
(S) TYPE:	(defaults to DATA ELEMENT)	N/A
(S) PROJECT:	(project name)	C25
(ML) DESCRIPTION:	(text description)	C60
(S) DATA TYPE:	(the type of data, if known)	C15
(S) MIN VALUE:	(minimum data value, if known)	C15
(S) MAX VALUE:	(maximum data value, if known)	C15
(S) RANGE:	(range of values, if applicable)	C60
(MF) VALUES:	(enumeration values, if appropriate)	C25
(MG) DECOMPOSITION:		N/A
(S) PART OF:	(parent data element name)	C25
(MF) COMPOSITION:	(subcomponent data element names)	C25
(MG) SOURCES/DESTINATIONS:		N/A
(MF) OUTPUTS:	(activity(s) where output)	C25
(MF) INPUTS:	(activity(s) where input)	C25
(MF) CONTROLS:	(activity(s) where a control)	C25
(ML) REFERENCE:	(cite a reference to the data element)	C60
(S) REF TYPE:	(the type of the reference)	C25
(S) VERSION:	(version of this data element)	C10
(ML) CHANGES:	(a history of the changes)	C60
(S) DATE:	(mm/dd/yy of this entry)	C8
(S) AUTHOR:	(author's name)	C25

Figure 21. Revised Data Dictionary Entry Format for Data Element

6. The VALUES field is lengthened to 25 characters to permit a wider range of allowable enumeration values.

7. The AUTHOR field is lengthened to 25 characters for compatibility with the activity data dictionary entry.

The Expert System Requirements

One of the purposes of this research is to demonstrate the feasibility of integrating an expert system with a CASE tool. In this case, it is the integration of the Ada based CASE tool, SAtool II, and an Ada based expert system that is used for the syntax checking of an $IDEF_0$ model.

As discussed in Chapter Two, the process of formalizing $IDEF_0$ syntax into predicate logic facts is outlined in (20). A prototype rule-based expert system is developed in (20) and expanded

in (24) to perform syntax checking of $IDEF_0$ syntax. The same Prolog expert system shell that represents $IDEF_0$ predicate logic facts as [Object, Attribute, Value] triples is used in both research efforts (20, 24). In addition, *if...then* constructs are used to represent the rules. By using a shell, only the rules and the facts are required - the inference engine is already supplied. Integration problems between the programming language C and Prolog plagued both research efforts.

Therefore, in order to maximize the reuse potential of past work while also avoiding past integration problems, the expert system selected for this research should have the following features:

- The expert system should, in fact, be an expert system tool (or shell).
- The expert system tool must be in Ada.
- The knowledge base of the expert system tool should have a procedural representation scheme. This means rules should be in the form of *if...then* constructs.
- The knowledge base and the working memory should have a fact representation scheme that allows facts to be stored as [Object, Attribute, Values] where 'Values' can be one or more values. This extension of the standard OAV triple is necessary, because the OAV triple cannot accurately represent some of the information in the essential data model that pertains to relationships.
- The expert system tool should have a clearly defined methodology for being embedded within Ada external programs.

The requirement for the expert system shell to be implemented in Ada severely constrained the available candidates. Several shells are discussed in (2), but all of them are commercial tools whose source code would probably not be made available. Ada source code for both forward and backward chaining expert system shells is presented in (4), but the code lacks a clearly defined methodology for being embedded within other Ada programs.

Early in this research investigation, an Ada version of CLIPS (CLIPS/Ada Version 4.3) that is fully syntax compatible with the C version of CLIPS was released by Computer Sciences Corporation under contract with the U.S. government. Fortunately, CLIPS/Ada meets all the above requirements. The following is an abbreviated list of its features:

- CLIPS/Ada is an expert system shell (or tool).
- CLIPS/Ada is implemented entirely in Ada.
- Rules in the knowledge base are in the form of *if...then* constructs as is shown in Chapter Two.
- Facts in working memory are represented in a LISP like format with one or more fields of data enclosed by parentheses. There is no maximum to the number of fields in a fact.
- The CLIPS/Ada Advanced Programming Guide details the ability of CLIPS/Ada to be integrated with external programs and its ability to be embedded within programs (32:2–24). These programs can be in several languages, including Ada.

In order to validate that CLIPS/Ada met the above requirements, a prototype expert system is developed and integrated with the OOD produced in (38). After modifying the source code in (38) to make it operational, the integration proceeded smoothly. Embedded calls to CLIPS from within Ada subprograms permitted facts to be loaded into working memory, a rule base to be loaded from disk, and CLIPS to be executed. A limited number of $IDEF_0$ syntactical features are examined by a similarly limited set of rules. Several test suites of $IDEF_0$ features are created and checked with no unexpected results.

Since CLIPS/Ada meets or exceeds all the necessary requirements, it is selected to become part of the Essential Subsystem and eventually part of SAtool II. Appendix C provides some additional information concerning CLIPS/Ada.

Summary

This chapter presents revised versions of the $IDEF_0$ Essential Data Model and the AFIT Data Dictionary Formats. Both models are provided as requirements documents for developing an Ada based object oriented design of the essential data model. However, several inconsistencies and inadequacies are identified in the models and are corrected prior to continuing with the object oriented design phase. The methodology used in identifying both the inconsistencies and the inadequacies is also presented. In addition, the rationale used in making the modifications to the models is also discussed. The revised models are then presented. The requirements for the expert system and a discussion of the selected expert system are presented as well.

Since an exhaustive review of the requirements is not a mandate for this research, no assertion is made that the revised $IDEF_0$ Essential Data Model and the revised AFIT Data Dictionary Formats presented in this chapter are free of additional inconsistencies, inadequacies, or other errors.

IV. DESIGN

Introduction

The beginning of the object oriented design process for the Essential Subsystem is presented in this chapter. Where applicable, the methodology used is discussed in terms of the object oriented design process methodology presented in (7).

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects. (7:191-195)

There are two basic modes to software design: functional design decomposition and object oriented design decomposition (39:185). The choice of an Ada based object oriented design decomposition approach over a more functional design decomposition approach is due to the desire to assess the feasibility of both Ada and the object oriented design process in the development of a CASE tool, SAtool II.

The choice of presenting the object oriented design process in terms of (7) is made because the methodology is both taught and advocated by AFIT/ENG, and it is AFIT/ENG students and faculty that will most likely perform follow on research to this investigation.

This primary focus of this chapter is on issues related to the first three steps of the object oriented design process. The next chapter includes information relevant to the last step in the process as well as other implementation details. Although the approach presented here may appear sequential at times, a highly iterative process actually occurred.

The first step of the object oriented design process consists of two tasks (7:123):

- Identify the classes and objects that form the vocabulary of the problem domain.

- Invent structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem.

The classes and objects are called *key abstractions* of the problem, and the cooperative structures are called the *mechanisms* of the implementation (7:123).

The first section in this chapter is a discussion of two opposing views of how to identify object classes. Although the discussion focuses on only the essential data model, the selected viewpoint influences the entire object oriented design process.

The key abstractions for the Essential Subsystem are derived by examining the four subproblems that are presented in Chapter Three. The mechanisms necessary for some of these object classes are then presented. Discussions on the output file formats and the expert system design are included as well. A graphical representation of the preliminary design is presented which is then followed by more detailed design information on the semantics and relationships among the objects of the Essential Subsystem. Based on this information, a more detailed graphical representation of the design is presented. Finally, a summary of the results of this chapter is presented.

The Level of Observation in Object Class Selection

The requirements specification presented in the last chapter states that the development and implementation of the essential data model are to be based on the corresponding E-R model. However, neither the requirements specification nor the subproblems based on those requirements specify from what level of observation the E-R model is to be viewed. A particular view is necessary since different object classes may be derived based on the level of observation the designer takes. Therefore, two different methods for identifying the object classes are considered. The methods considered are based on the level of observation taken towards an $IDEF_0$ model: an outside (public) view or an inside (private) view. Note that the inside and outside views discussed here are not intended to correspond to the inside and outside views discussed by Booch (7:123).

A completed $IDEF_0$ model viewed from the outside is a hierarchy of $IDEF_0$ diagrams with the A-0 diagram at the top of the hierarchy. This hierarchy can be modeled as a tree where each node of the tree is an individual $IDEF_0$ diagram. Incomplete $IDEF_0$ models that are built from the bottom up can then be viewed as a forest of trees.

A single, complete $IDEF_0$ diagram viewed from the outside closely resembles a graph with the exception that there is no apparent "source" for the incoming arcs and no apparent "sink" for the outgoing arcs. Figure 22 illustrates how the $IDEF_0$ diagram of Figure 10 could be modeled as a graph by adding a "source activity" and a "sink activity". However, note the introduction of the small circles that model the branching of data elements. The places where arrows meet are sometimes referred to as *junctors* (14:61) and would have to be modeled somehow in a graph representation scheme.

From the outside, entities such as activities and data elements would simply be nested objects or subcomponents of the "node" object class, where node represents an $IDEF_0$ diagram. Relationships between activities, data elements and other entities could be embedded within the different object class definitions. For example, the relationship 'composed of' could be correctly modeled by the tree. Intuitively, each branch of a tree could be modeled as one or more instances of the relationship 'composed of' since the branches represent parent-child relationship among the data in a tree.

An inside view of an $IDEF_0$ model is that of a single $IDEF_0$ diagram. The inside view does not explicitly represent the hierarchy of diagrams or the diagrams themselves. Its primary concern lies solely with the entities and their relationships. However, the hierarchy of diagrams can be determined, as well as the activities and data elements that appear on a diagram. These entities are not explicitly modeled as they are in the outside view. This appears to be the intended viewpoint of the $IDEF_0$ Essential Data Model presented in the previous chapter. The essential data model does not contain a *hierarchy* entity nor a *diagram* entity, since that information can

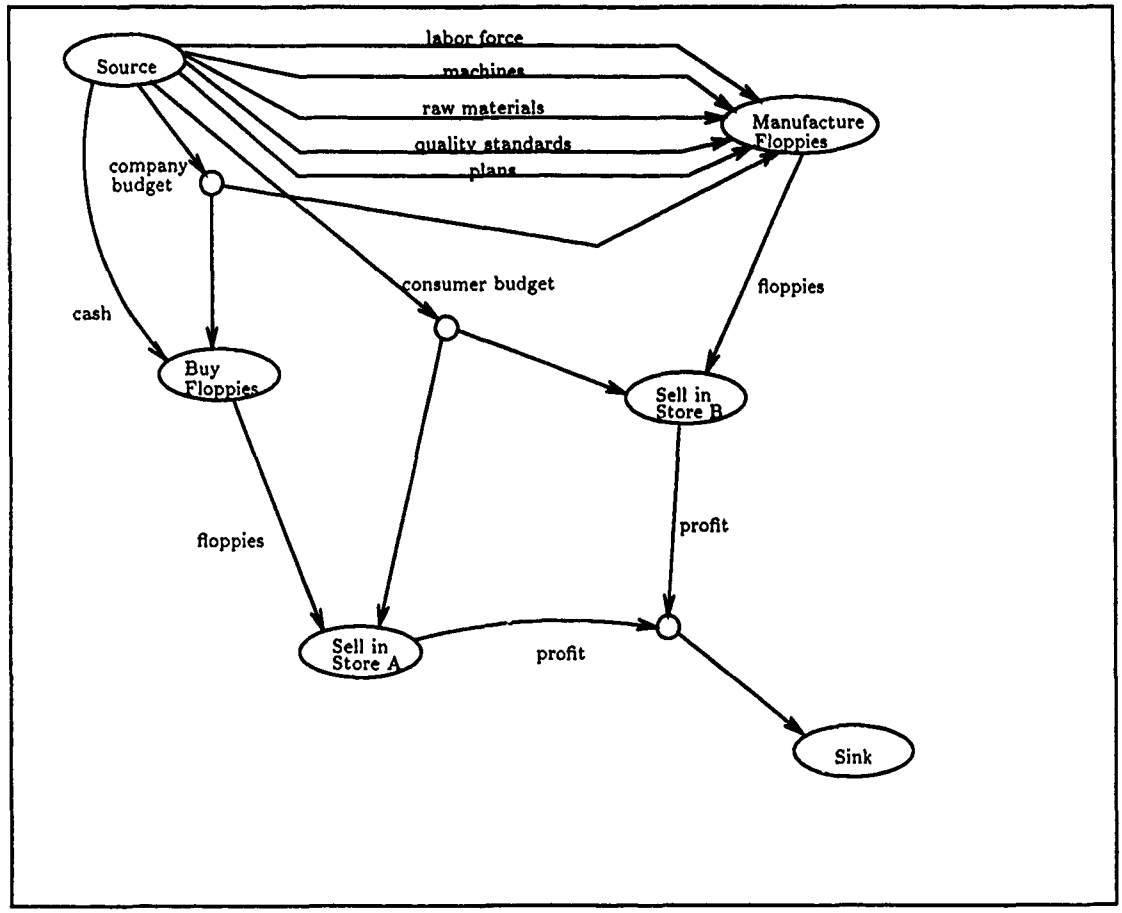


Figure 22. Graph Representation of $IDEF_0$ Diagram for 'Market Floppies'

be derived from the relationships. The hierarchy, for example, is modeled as a single relationship called 'composed of'. Note that in the outside view, the hierarchy is the entire model, whereas in the inside view, it is just one E-R construct among many others.

There are several reasons for selecting the inside view of the $IDEF_0$ model as the level of observation from which the objects are to be identified. Some of the reasons are influenced by implementation issues. Theoretically, implementation issues should not be addressed at this point in the object oriented design process, but considering these issues is sometimes unavoidable.

The primary reason for choosing the inside view is flexibility. The object classes necessary to implement the outside view would require relationships imbedded within them. By imbedding relationships within object classes, the number of objects implemented can be reduced. However, this can result in tighter coupling between objects. Thus, adding or changing relationships would be both difficult and time consuming. Explicit modeling of relationships in the inside view, however, allows for greater ease in adding and modifying relationships at a later point in time. An example of the lack of flexibility in an OOD is provided by Smith's OOD for the essential data model which consists of only three object classes (38:4–14). His design strategy resulted in all relationships being embedded as subcomponents of object classes.

As shown in Figure 22, the most likely way to model an $IDEF_0$ diagram from the outside view is a graph. Unfortunately, a graph cannot sufficiently model all the different relationships between activities and data elements regardless of how the graph is actually implemented. An arc in a graph consists of a pair of vertices (18:273) just as a data element relates two activities. However, when an arrow connects two boxes, there are additional relationships based on where the arrow physically connects to the boxes. A graph has no equivalent paradigm for these additional relationships. Since an arc must consist of two vertices, a graph also cannot model an arrow without a source or destination which would be a common occurrence in a CASE tool based on $IDEF_0$. A graph can consist of partial graphs that may only be a single node, but all arcs in a graph must

consist of a pair of vertices (18:273). Also of concern are the junctors. In a graph where activities are modeled as the nodes of the graph, would the junctors also be modeled as nodes?

An examination of the terminology used in the discussion of the opposing views highlights another, more subtle reason. In discussing the outside view, the terms used are *hierarchy of diagrams* and *diagram*. These terms are more closely related to graphical representation of an $IDEF_0$ model. Thus, the outside view appears to be more closely oriented with the drawing data model view of the $IDEF_0$ language.

E-R Model to OOD Mapping Technique

Due, in part, to the size and complexity of the $IDEF_0$ Essential Data Model, the process of identifying objects from the E-R models demands a systematic approach. Therefore, a mapping technique is developed from the entities and relationships in an E-R model to OOD object classes. Ideas from both the *Keystone System Design Methodology* (23) and *Object-Oriented System Analysis* (37) are used in developing the technique and defining the object classes.

The concept that all relationships in an E-R should be modeled as objects is considered from (23:102). In fact, this concept is partially adopted. Concepts from (37) are used in determining how to model relationships as objects.

The mapping technique presented here is the result of a joint effort (40) and can be used to map the entities and relationships of any E-R model containing only those E-R constructs that appear in the essential data model. Hence, it is not a generalized mapping technique for all possible E-R models. The three types of relationships present in the essential data model are referred to in terms of their cardinality with the entities they relate. Note that the relationships listed below implicitly include their conditional variations. For example, the 'many to many' relationship implicitly includes the 'many to many conditional' relationship. The conditional variations mentioned here correspond to those presented in (37:60–64).

- one to one relationships
- one to many relationships (including many to one)
- many to many relationships

It is the above relationships that are used to drive the object selection process. The following steps detail the mapping technique used.

1. Any entity that participates in more than one relationship becomes an object class.
2. Any 'many to many' relationship that relates two entities already identified as object classes becomes an object class itself.
3. Multiple 'many to many' relationships between two entities which have already been mapped into object classes are combined into a single object class only if the relationships they model between the two entities are similar. If dissimilar, they remain as separate objects.
4. An entity (including its attributes) that participates in a single relationship that is a 'many to many' relationship becomes a multiple field attribute of the second entity in the relationship. The 'many to many' relationship is not mapped into an object class, and any attributes of the relationship become attributes of the other entity as well. Note that the first entity, in this case, should not be "complex", where complexity is defined as an entity possessing at least one multiple field attribute. If the entity is complex, then the entity, as well as the relationship, should be modeled as objects.
5. An entity (including any attributes) that participates in a single relationship that is either 'one to one' or 'one to many' becomes either a single field or multi-field attribute of the other entity. The relationship is not mapped into an object class, and any attributes of the relationship become attributes of the other entity as well. Again, note that the entity, in this case, should not be "complex". If the entity is complex, the entity is modeled as an object,

and the relationship is modeled by placing a key attribute from one of the entities into the other entity as a foreign key.

6. Any remaining relationships are either 'one to one' or 'one to many' relationships between entities that have already been mapped to object classes. These relationships are not mapped into object classes. For the 'one to one' relationship, a foreign key attribute of either of the entities must become an attribute of the other entity. For 'one to many' relationships, a foreign key in the many entity became a multi-field attribute of the one entity.

The Key Abstractions

This section presents the key abstractions of the problem and solution space.

Essential Data Model Key Abstractions. Applying the mapping technique to the essential data model yields the object classes and associated attributes depicted in Table 2. All attributes in the table consist of a single field of data unless otherwise noted.

Upon examination of the object classes presented in Table 2, it is clear that the mapping technique presented earlier is not followed in all cases. The following exceptions to the mapping technique are made:

- Since the 'reference' entity will likely be infrequently used, mapping it and the two 'based on' relationships into objects seem unnecessary. Therefore, the reference entity is considered to be two separate entities – one related to an activity and one related to a data element. By abstracting 'reference' as two separate entities, step four in the mapping technique is applied to permit them to be modeled as attributes. Furthermore, the cardinality between the activity entity and the reference entity was reduced for implementation purposes down to a 'one to m' versus a 'n to m' relationship. Thus, an activity is allowed only one reference, where a reference consists of a multi-line 'reference' field and a single 'reference type' field.

Table 2. Objects Classes and Attributes Based on the Essential Data Model

Object Class	Attributes
Project	Name
Activity	Name Number Description (multiple lines, single field) Children (multiple fields) Reference (multiple lines, single field) Reference Type Version Changes (multiple lines, single field) Date Author
Data Element	Name Data Type Minimum Maximum Data Range Values (multiple fields) Description (multiple lines, single field) Reference (multiple lines, single field) Reference Type Version Changes (multiple lines, single field) Date Author
Historical Activity	Project Name Activity Number
Calls Relation	Activity Name Historical Activity
Consists Of Relation	Decomposition Id Parent Activity Name Child Activity Name
ICOM Relation	ICOM Id Activity Name Data Element Name Relationship Type

- Since the implementation of SAtool II will only permit a single project, there is no need to model the relationships between 'project' and 'data element' and 'project' and 'activity'. By default, all activities and data elements that exist in the model belong to the single project.

Each of the object classes in Table 2 models one or more of the entities and relationships depicted in the $IDEF_0$ Essential Data Model.

1. Activity Class. This class models the entity 'activity' and its attributes. It also models the relationship 'composed of' via the multi-field attribute 'children' which contains a list of the activity's child activities (if any). The activity class attributes 'reference' and 'reference type' model their respective attributes of the 'reference' entity in the E-R model. This modeling is possible because of the aforementioned exceptions made to the mapping technique. An instance of this class (i.e., an object) is uniquely identified by the 'name' attribute.
2. Data Element Class. This class models the entity 'data element' and its attributes. Its 'reference' and 'reference type' attributes model their respective counterparts in the data element essential data model. Objects in this class are uniquely identified by the 'name' attribute.
3. Historical Activity Class. This class models the entity 'historical activity' and its attributes. Note that the entity 'historical activity', like the entities 'activity' and 'data element', appears in both the essential and drawing data models. Objects in this class are uniquely identified by a combination of the 'project name' and 'activity number'.
4. Calls Relation Class. This class models the relationship 'calls' which relates an activity to one or more historical activities. Objects in this class are uniquely identified by a combination of an 'activity name' and the key from the historical activity class – 'historical activity'.
5. Consists Of Relation Class. This class models the relationship 'consists of'. For every one data element that is decomposed, two or more objects in this class are created based upon

how many subcomponents the data element is split into. For example, if the data element 'm' is decomposed into 'a' and 'b', two objects from the consists of class would be created and assigned the same "decomposition id". The "decomposition id" insures that if 'm' is decomposed in a different manner elsewhere in the model, that decomposition can be differentiated from another decomposition of 'm'. This capability is necessary to correctly model the $IDEF_0$ feature that permits a data element to appear in multiple decompositions (i.e., have multiple parents). Objects in this class are uniquely identified by a combination of all its attributes depicted in Table 2. Note that a decomposition of a data element can be derived not only from the data element being "split" into two or more parts but also from two or more data elements being "joined" together.

Replications of data elements are not members of this class. For example, if data element 'm' is split into 'm' and 'm', this is considered a *replication* of 'm' and not a decomposition. Replications of data elements decrease the clutter on an $IDEF_0$ diagram by reducing the number of arrows required between boxes but have no meaning in the essential data model.

Consistency within this class is maintained by considering each data element as a set. A data element that does not consist of any other data elements is a set with one member – itself. Otherwise, a data element is a set containing its component data elements. Thus, if performing the set theory operation *union* among all the subcomponents a data element yields the same data element, consistency is achieved. The application of the union operator among the subcomponents is not without precedence since it is also used by Ross (36:20). It is this view of the Consists Of Class that is enforced in the implementation of the Consists Of Relation Manager presented in the next chapter.

6. ICOM Relation Class. This single class models the four relationships that exist between activities and data elements. The "relationship type" attribute captures the different relationships by having the values 'i', 'c', 'o', or 'm' which represents input, control, output, and

Table 3. ICOM Relation Class Instances for Data Element 'floppies'.

ICOM Id	Activity	Data Element	Relationship
1	Buy Floppies	Floppies	o
1	Sell in store A	Floppies	i
2	Manufacture Floppies	Floppies	o
2	Sell in store B	Floppies	i

mechanism respectively. An object of this class is created whenever a data element (an arrow) is related to (is connected to) an activity (a box). The "ICOM id" is necessary to handle the multiple source-destination groupings that can occur. For example, Table 3 illustrates the four instances of the ICOM Relation Class that would be required to correctly model the data element 'floppies' in Figure 10.

Note that the term "ICOM" used in this context should not be confused with the ICOM codes discussed in the $IDEF_0$ manual (30:4-8).

The Data Dictionary. The discussion presented here refers to the identification of any object classes related to the creation, editing, or output of data dictionaries. The data dictionary used by SAtool II is simply a human readable representation of the essential data model information. The creation of a data dictionary entry is considered to be the creation of ASCII text that contains the fields of a data dictionary entry with the appropriate information from the essential data model displayed in the fields. Thus, at least one object class is required to model the text representation of the data dictionary entry. To output a data dictionary entry to a file or to output device, the same object class can be used.

Editing a data dictionary entry is another part of the problem. As stated earlier, the data dictionary is a human readable representation for the information in the essential data model. The data dictionary can, therefore, be used as a template for editing $IDEF_0$ features not shown on the $IDEF_0$ diagram. For example, a data element has a version number, but the $IDEF_0$ language has

no way to represent that version number on the $IDEF_0$ diagram itself. Thus, the individual fields of the data element data dictionary entry format are used as a convenient human-machine interface for entering the essential data model information. The key abstractions that capture the user input are part of the graphical user interface. Therefore, they are not modeled here.

Consequently, only a single object class, *Data Dictionary* is identified. The format for storing data dictionary information is explained in a later section within this chapter.

The CLIPS/Ada Expert System Interface. There are two options for integrating CLIPS/Ada with SAtool II:

1. Embed CLIPS/Ada within SAtool II. SAtool II is therefore the client procedure.
2. Make calls to SAtool II from within CLIPS/Ada. CLIPS/Ada is therefore the client procedure.

The first option is clearly the desired one, since the objective is a CASE tool that can perform syntax checking via an expert system, not the reverse.

As with most expert system shells, CLIPS/Ada already provides an inference engine. In fact, CLIPS/Ada employs a forward chaining reasoning method (35:128). The two remaining components are the knowledge base and the working memory. The knowledge base in this case contains the rules for $IDEF_0$ syntax checking. These rules, together with facts about the problem to be solved, are loaded into the CLIPS/Ada working memory where the inference engine can employ the three phases of the recognize-act cycle.

In this case, an interface to the CLIPS working memory is needed, so a client program (SAtool II) can add essential data model facts and load rules from the knowledge base into the working memory. Therefore, a key abstraction called *CLIPS Working Memory Interface* is suggested to model the interface between the CLIPS working memory and SAtool II.

The Essential Data Model Information. In order to save and restore the state of an $IDEF_0$ model, the key abstraction *Essential IO* is identified. To physically store the information that is contained in instances of the object classes of the essential data model, an interface to the system is required to output a file. Likewise, when loading essential data model information, a file of essential data model information is required. Essential IO models that interface.

The issue of storing essential data model information to a file triggers a question. What is the format of the information to be output? The answer to this question is presented in the *File Formats* section.

Error Handler. The *basic strategy* for exception/error handling (25:91) accurately desc[illegible] the initial methodology used in this research . However, this strategy is considered inadequa[illegible] large, object-based Ada systems for the following reasons (25:93–94):

- Exception handlers are replicated.
- Maintenance of exception handlers is difficult.
- Limiting and controlling error reporting is difficult.

Although the term "large" is never quantified (25), the above difficulties accurately reflect the problems in implementing the initial error handling methodology used for this research. Several complex alternative methods are presented for overcoming these difficulties (25:94–101). However, implementing any of these alternatives requires significantly more time than is available for this research. Therefore, an alternative, less complex, error handling methodology is jointly devised (40). In short, the basic strategy is abandoned in lieu of a more centralized approach to error handling. Since the design methodology for this research is object oriented, the new error handling methodology is modeled as another key abstraction, the *Error Handler*, whose purpose is to provide a single focal point in the model for error handling.

The Mechanisms

Now that the key abstractions or object classes are identified, the mechanisms by which they interact must also be defined. It is these mechanisms that are considered the *soul* of the design.

> Whereas key abstractions reflect the vocabulary of the problem domain, mechanisms are the soul of the design. During the design process, the developer must consider not only the design of individual classes, but also how instances of the classes work together (7:148).

It is instances of classes working together that drives the identification of the mechanisms that appear in this section.

During the creation of an $IDEF_0$ model, multiple instances of each of the object classes derived from the essential data model occur. An $IDEF_0$ model consists of multiple instances of the data element class, the activity class, etc. Multiple instances of the same object class are grouped together into homogeneous collections to properly reflect their interrelationship with one another. Of all the object classes identified so far, only the object classes derived from the essential data model have multiple instances.

By examining Table 2, it appears that seven different mechanisms or *manager mechanisms* will be required. However, the current implementation of SAtool II permits only one instance of the project class. Therefore, no mechanism for the project class is necessary. This leaves the following six manager mechanisms:

- Activity_Manager
- Data_Element_Manager
- Consists_Of_Relation_Manager
- Historical_Activity_Manager
- Calls_Relation_Manager

- ICOM_Relation_Manager

There is some underlying similarity in these six manager mechanisms. The apparent function of each one is simply to manage multiple instances of a class. Therefore, the concept of a *generic manager* is suggested.

> Another name for a generic manager is a *parameterized class.*

> A *parameterized class* (also known as a generic class) is one that serves as a template for other classes – a template that may be parameterized by other classes, objects, and/or operations (7:118).

By labeling the generic class as a class in (7), it implies it is a key abstraction. However, the same construct is later referred to as a generic class mechanism (7:118). Thus, it is not clear whether the generic class is considered a class, a mechanism, or some combination thereof. For the purpose of this research, however, the terms generic class and/or parameterized class are considered as key abstractions. Consequently, an additional key abstraction is identified during the process of identifying mechanisms. The term *Generic Multiple Object Manager* is used to represent this key abstraction.

File Formats

Having identified the *Essential IO* and *Data Dictionary* as key abstractions that output information, a discussion of the output format for both the data dictionary and the essential data model is necessary[1].

The output format for data dictionary information is based on the formats presented in Tables 20 and 21 in Chapter Three. The textual description of the fields, the character lengths of the fields, and the field classifications (i.e., (S), (G), (M), (MF), (ML) and (MG)) are not included

[1] Additional SAtool II output file formats can be found in (40).

in the output format. Any file name associated with an output data dictionary is affixed with an '.edd' extension.

The output format for the essential data model (sometimes referred to as the project data) is significantly different from the data dictionary. Actually, since the data dictionary is suppose to represent all the essential data model information, the data dictionary format could theoretically be used as the output format. The original intent of this research was to follow the file format established in (11). However, an examination of the format revealed that it is based on the earlier interpretation of the $IDEF_0$ language that did not account for multiple decomposition of data elements or multiple source-destination groupings of data elements. Thus, to use this format, changes in the format itself would be necessary. Since research time is not allocated for such changes, an alternative method is devised.

The requirement for the expert system to perform syntactical checks of the $IDEF_0$ syntax meant that a formalization of the $IDEF_0$ syntax into a 'computer readable' form would be required. As discussed in Chapter Two, the formalization of the process of transforming $IDEF_0$ syntax into predicate logic facts a. ıen to OAV triples for use in an expert system is already accomplished (20). Because CLIPS facts have a similar representation scheme, the process can be readily adapted to converting the information in the essential data model to CLIPS facts. These facts can then be transferred to the *CLIPS Working Memory Interface.*

With one formalization method already implemented (20), a joint decision (40) is made to simply redirect the facts from the CLIPS Working Memory Interface to an output file whenever a project is to be stored. Therefore, the file format for the essential data model is simply that of a CLIPS fact file. The same file format could also be applied to the drawing data model information which is stored separately (40). Output files with essential data model information carry an '.esm' extension. Output files with drawing model information carry a '.drm' extension. More information on fact representation is presented in the next section.

Expert System

There are three main concerns for the successful integration of the expert system with the Essential Subsystem and eventually SAtool II:

- The interface between the Essential Subsystem (SAtool II) and CLIPS/Ada.
- The representation and formulation of the facts.
- The representation and formulation of the rules.

Since the interface is achieved by the CLIPS Working Memory Interface package, this section focuses on the last two concerns.

Note that the expert system design process is not presented as a separate methodology in this research but is integrated within the Essential Subsystem design process. Comprehensive expert system development methodologies do exist (21, 22) but do not apply in this case, since the primary goal is the creation of a CASE tool that uses an expert system – not an expert system that uses a CASE tool.

As stated in the last chapter, the representation method for the essential data model information is CLIPS/Ada facts. Facts are used to represent the essential data model when it is being stored to a file and when it is to be loaded into the CLIPS working memory for syntactical checks. Facts in CLIPS/Ada have a "LISP-like" notation of one or more fields surrounded by a single set of parentheses (35:2). A field can be either a word, a number, or a string, and are totally free form, i.e., the only limit on the number of fields in a fact is the memory of the system (34:I-2–I-8). Previous research efforts (20, 24) use only facts of the form [Object, Attribute, Value] which are otherwise known as OAV triples. These investigations conclude that OAV triples are sufficient for modeling a subset of the essential data model information.

Unfortunately, those investigations (20, 24) have two deficiencies:

1. Neither investigation translates every feature of an $IDEF_0$ model into facts.

2. Both investigations use a paradigm of the essential data model that does not correctly model multiple decompositions of data elements and multiple source-destination groupings of data elements.

As a consequence of correcting the first deficiency, the OAV construct for representing the essential data model information is no longer adequate. For example, both activities and data elements have multi-line descriptions, where each line consists of a number of words within a 60 character limit. If the OAV triple construct is used, a fact for each word must be declared. On the other hand, if the fact representation method is revised, a single fact can represent all the words in a single line of the description. Thus,

(act-desc "activity name" word1 word2 word3 ...)

is a CLIPS fact which has the general form (attribute, object, value, value, ...), where 'attribute' defines the type of fact, 'object' is a variable that is the actual name of the activity, and 'value' is a word of the description. Note that CLIPS does not care how the information in a fact is arranged, it only cares that the information is enclosed in parentheses and separated by one or more spaces. Therefore, the user is free to implement any positional representation scheme as long as it in a LISP-like format.

The addition of the 'Consists Of Relation Manager' and the 'ICOM Relation Manager' corrects the second deficiency but exposes another inadequacy in the OAV construct. By modeling relations as objects, the OAV triple is again no longer adequate. For example, the ICOM information presented in Table 3 models a relationship between two objects – a data element and an activity. The relationship between two objects does not fit into the OAV triple construct which has but one object in its tuple. Therefore, a revised methodology for representing the information in relationships is necessary. Basically, the method chosen is an attribute followed by the tuple of the relationship. Thus,

(icom-tuple Buy_Floppies Floppies o 1)

is the CLIPS fact representing the first entry in Table 3. This same methodology is applied to the representation of facts for each of the objects that implement relationships.

Appendix G presents examples of the facts that are sent to the CLIPS Working Memory Interface object and Appendix F presents examples of facts that are used to store the state of the essential data model. Of particular interest is that there are two different groups of facts: syntactical facts, and state representation facts. Syntactical facts are those sent to the CLIPS working memory for syntax checking, whereas state representation facts simply represent the state of the $IDEF_0$ model. This difference is evident in the representation of description of an activity or data element. From a syntactical viewpoint, the value of the description is irrelevant; its presence or absence is the only concern. Therefore, the facts

(act-desc "activity name" not-null)

and

(act-desc "activity name" null)

represent the presence and the absence of an activity description respectively. Only one fact or the other is placed into the working memory of the CLIPS expert system. On the other hand, the state representation of the description requires the "value" of the description to be stored. Thus, its fact representation is that shown earlier in this section.

As shown in Chapter Two, the rules of a CLIPS/Ada expert system are in the form of *if then* constructs. For example, the rule below prints a warning message if an activity is found without a description. The '?act' acts as a variable which is instantiated with an activity name if the rest of the left hand side of the rule is matched as well. Thus, every activity with a null description triggers a warning message to the user.

```
(defrule null-activity-description
 (act-desc ?act null)
 =>
 (printout t "Warning: Activity" ?act "needs a description." crlf))
```

Many rules, similar to the above rule, are to be written. The purpose of the rules is to check the syntactic validity of an $IDEF_0$ model. These are just a few of the checks that the expert system should perform.

1. The number of input, output, control, and mechanism arrows for each activity should be checked to insure a valid number of them are in place.
2. The boundary arrows of an activity and its parent activity should be compared to insure the $IDEF_0$ hierarchy is consistent.
3. The description and activity number of an activity should be checked for their presence or absence.

The next chapter describes the rules that are actually implemented.

Preliminary Design

Figure 23 is an *object diagram* (7:171) of the preliminary design. The notation used is based on (7:170). Note that the CLIPS/Ada expert system is represented by the object *CLIPS* in Figure 23.

The Generic Multiple Object Manager

Because the Generic Multiple Object Manager is the building block for many of the objects a more detailed discussion of its design is warranted. The term *encapsulation* used in this discussion means that the inside view of an object class is hidden from all client programs. In other words, client programs can only access an instance of the object through a selected set of operations or *methods*. It is only through these operations that an instance of an object class can be acted upon by client programs.

The recognition of mechanisms as the *soul* of the design (7:148) is particularly relevant to the generic class Generic Multiple Object Manager. The design and implementation of this generic

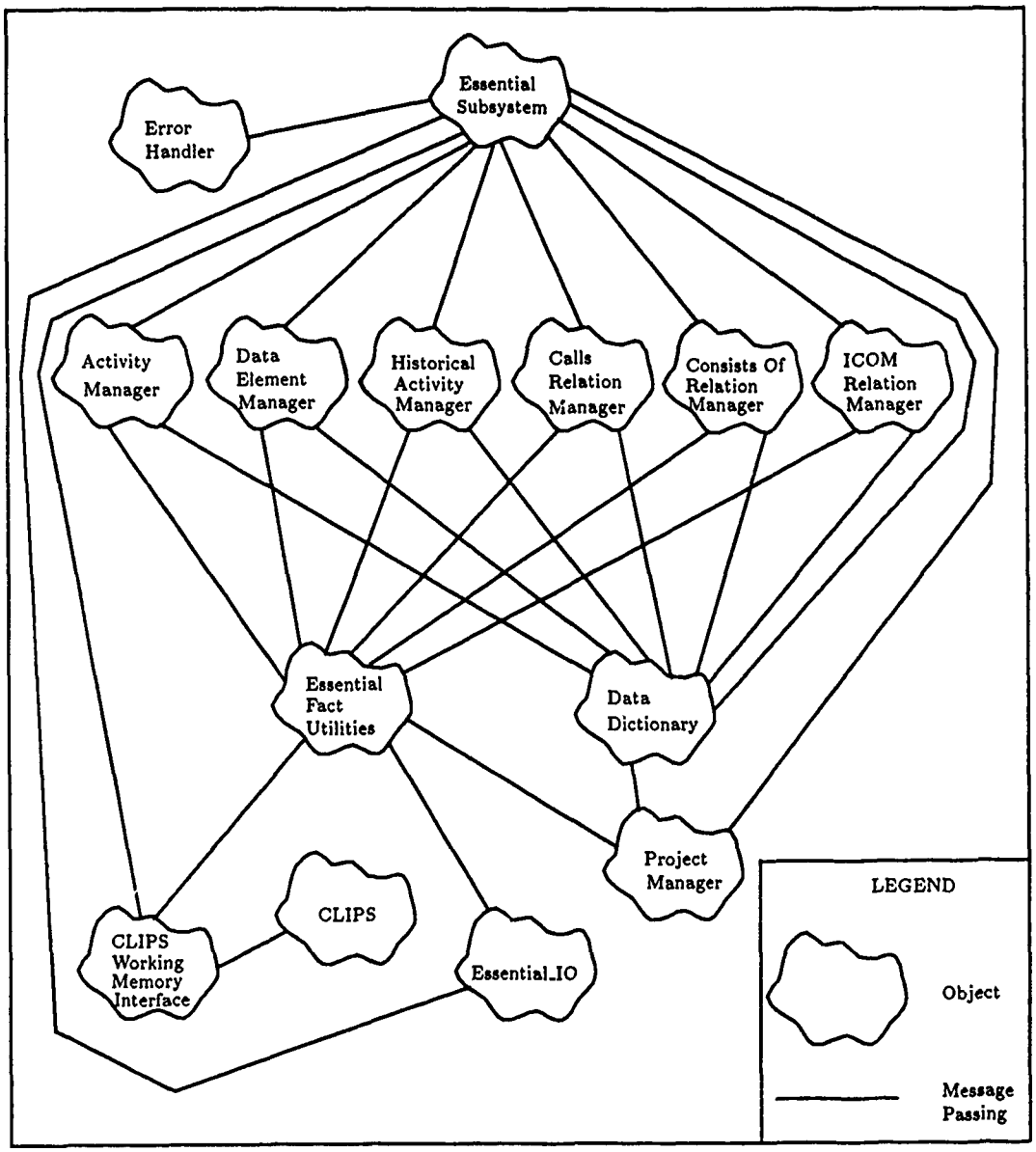


Figure 23. Object Diagram Depicting Preliminary Essential Subsystem Design

class either directly or indirectly impacts on almost all other objects in the Essential Subsystem and the rest of SAtool II.

In both the essential data model and drawing data model, there exists a requirement for a mechanism for managing multiple instances of object classes. However, some of the object classes have multiple attribute fields. If strict encapsulation of these classes is followed, three layers of abstraction may be implemented between the client program and an instance of the object class.

- Layer 1. The object class itself is encapsulated with one or more operations. Typically, there are at least two operations for each attribute – one to set the value of an attribute and one to determine the value of an attribute. Operations to create, initialize, and delete the object are also typically implemented.
- Layer 2. The instantiation of the generic class to manage instances of a particular object class results in an object class – not an object. An instance of this object class results in the actual manager mechanism which now allows primitive operations on the manager to be performed. These normally include is_empty, add_item, remove_item, initialize, clear, value_of_item, etc. However, to actually access the internal representation of the item, the operations imposed by the Layer 1 level of abstraction must be used.
- Layer 3. Once the client programs within SAtool II need to perform more than just the primitive operations available from the generic class, the manager mechanism must also be encapsulated which results in a third layer of abstraction.

Encapsulation of object classes has its consequences in terms of increased system overhead because of additional message passing (7:216). Since the response time of a CASE tool is critical to its acceptance by the user community, the decision is made to eliminate a layer of abstraction by not encapsulating any of the object classes from the essential data model[2]. This decision also reduces the size of the object code.

[2]This applies to those object classes with multiple instances which excludes the project class.

In order to increase the efficiency of the Generic Multiple Object Manager itself, a unique design decision is made. What makes the design and implementation of this generic class unique is that it exports an Ada access type (i.e., a pointer) that permits direct access to any item in the manager. This implementation was considered to be necessary because of the presence of object classes with large number of attributes. Operations to change these attributes typically take O(the_manager_size) time[3] in the worst case for unordered items. This is considered unacceptable for a CASE tool. Therefore, the risks involved in violating the encapsulation philosophy are accepted and implemented. This implementation thus reduces the time complexity of the operations on the attributes to O(1) time.

Fortunately, this generic class can also be used to manage multiple instances of object classes with only a single field (i.e., an object class with a single type). Several examples are presented in the discussion of the Environment_Types package in the next section. A demonstration of the flexibility of the Generic Multiple Object Manager is offered by an examination of the source code contained in Volume II of this research which reveals no other mechanisms for the management of multiple instances of object classes (e.g., linked lists, queues, rings, etc.).

The Semantics

The semantics of an object are the operations (or methods) that client programs can perform on the object and the operations that the object performs to act upon other objects (7:80). In this research, the methods for each of the objects are represented via Ada package specifications. It is these specifications, viewed together as a whole, that describe the semantics of all the objects in the Essential Subsystem.

[3]For an explanation of the Big-O notation see the Order Of Analysis section in the next chapter.

The semantics (i.e., the package specifications) for each of the objects can be found in Volume II of this research which contains all the source code. Volume II is available through the Air Force Institute of Technology, Department of Electrical and Computer Engineering.

Note that the Activity Manager, like all the other managers, is a server object (7:89). In other words, the managers have only suffered operations upon them and do not operate on other objects in the subsystem. The reason for this design decision is explained in the next section.

The Relationships and Visibilities

This section discusses the reasoning behind the relationships and visibilities among the various objects in the Essential Subsystem. A *relationship* between two objects means that message passing occurs between them (7:170). The *visibility* between two objects details *how* the two objects see one another.

Of particular interest are the relationships among the six manager mechanisms. The issue is one of responsibility for the integrity of the design. In other words, are the manager mechanisms themselves responsible for updating and maintaining consistency among themselves, or is that the responsibility of a client program? These integrity requirements are called *integrity constraints* (37:53). An example of an integrity constraint is that every data element appearing in the Consists Of Relation Manager mechanism must also appear as a member of the Data Element Manager mechanism. If that is not the case, an integrity constraint is violated.

The decision to require the client program to enforce the integrity constraints is made for two reasons: reduced coupling and greater flexibility. By requiring a client program to enforce these integrity constraints, there is no requirement for any visibility between the manager mechanisms. This reduces the mechanism coupling, but it also increases the complexity of the client program. Greater flexibility means that one or more additional relationships and/or entities can be added

to the essential data model without requiring modifications to the other six manager mechanisms. Thus, the end result is no coupling among the six manager mechanisms.

Figure 23 illustrates the relationships among the objects of the Essential Subsystem by showing which objects communicate with one another via message passing. However, in order to classify how each of the objects see one another (i.e., their visibility), it is necessary to add additional detail to Figure 23. Therefore, Figure 24 illustrates a more detailed object diagram that includes not only the relationships among the objects but also their visibilities.

Summary

This chapter presents the first three steps in the object oriented design process. The concepts of key abstractions and mechanisms presented in (7) are extremely helpful in the proper classification of the objects of the problem domain. Seven object classes based on the essential data model are identified. In addition, object classes based on the data dictionary, the essential data model (project) information, the interface to CLIPS, and error handling are also identified.

The mechanisms necessary for the management of multiple instances of similar classes are discussed. Six different mechanisms are identified. The iterative nature of the design process is reflected in two ways:

- The identification of the key abstraction *Generic Multiple Object Manager* during the process of identifying the mechanisms for the essential data model.
- The identification of the key abstraction *Error Handler* during the implementation phase.

The output formats for both the essential data model information and the data dictionaries are also presented.

The design of the expert system is also discussed. The state information of an $IDEF_0$ model is stored and restored in the form of CLIPS/Ada facts. Where past research efforts used only

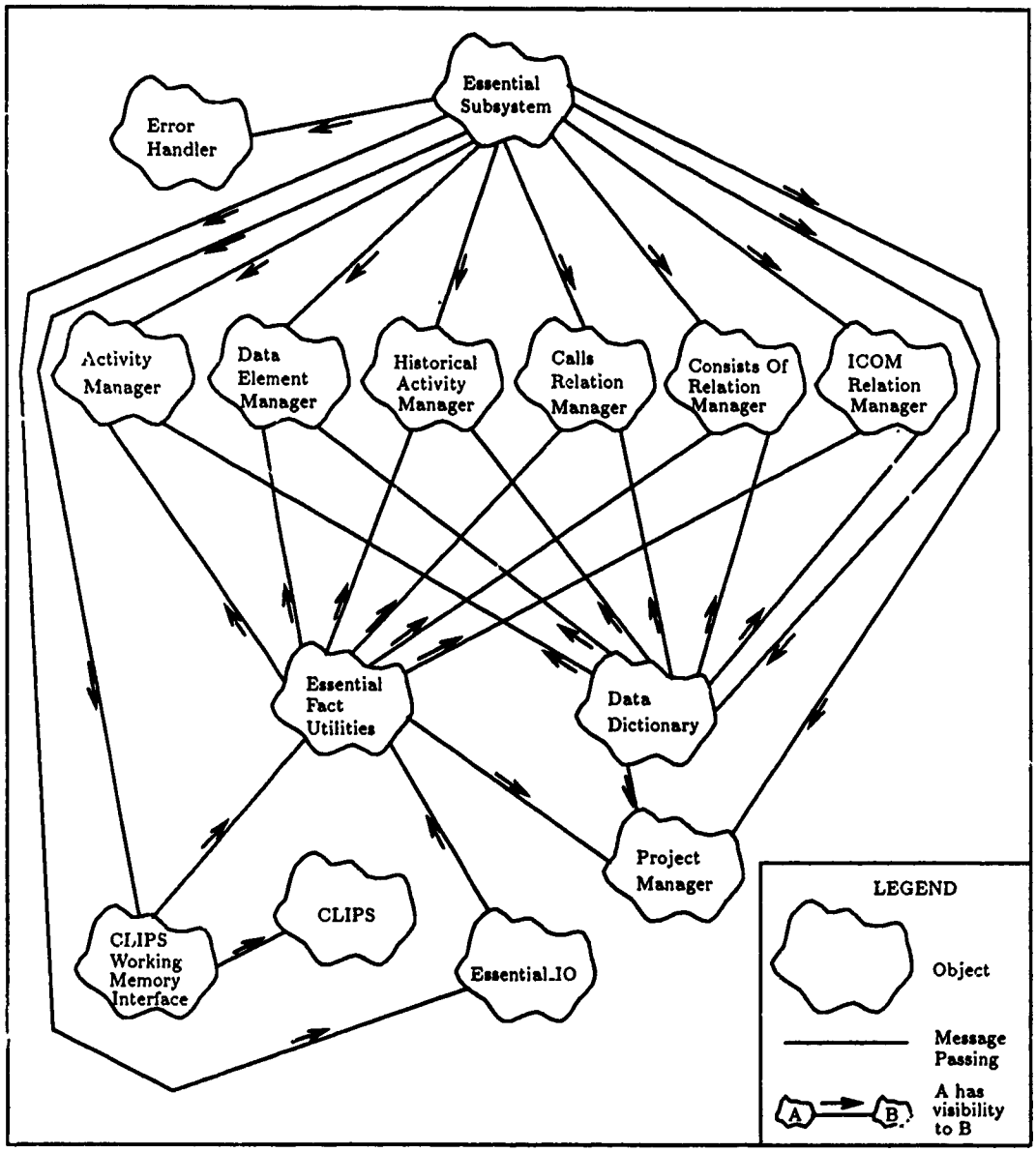


Figure 24. Essential Subsystem Detailed Design

OAV triples (20, 24), this research finds that representation insufficient and develops new fact representation methods.

The interface to the CLIPS/Ada expert system is modeled by the CLIPS Working Memory Interface object. This object uses the operations of the Essential_Fact_Utilities object to obtain the facts pertaining to the syntactical make up of an $IDEF_0$ model and then transfers them to the working memory of the CLIPS expert system.

A preliminary design is devised based on the functionality of the objects. The rationale used in determining the design of the Generic Multiple Object Manager is then discussed. Of special note is the capability of the manager to permit direct access to an item within the manager.

After discussing the visibilities and relationships among the objects, a more detailed design is then presented. Coupling among the manager mechanisms is eliminated by placing the responsibility for integrity constraints onto the client program.

V. IMPLEMENTATION, TESTING, AND INTEGRATION

Introduction

This chapter presents the implementation, and testing of the *Essential Subsystem* which includes the CLIPS/Ada expert system.

The first section discusses the implementation of each of the packages which model the objects of the Essential Subsystem. This is followed by a discussion of the expert system implementation. The documentation standards and order-of analysis methodology are presented next. Finally, testing and a limited discussion of the integration of the Essential Subsystem with SAtool II are presented.

The Essential Subsystem Packages

In this section, the implementation for each package that is part of the Essential Subsystem is discussed. The physical design of the Essential Subsystem is graphically depicted using *module diagrams.*

> A module diagram is used to show the allocation of classes and objects to modules in the physical design of a system; a single module diagram represents all or part of the module architecture of a system (7:175).

The module diagram of Figure 25 represents the architecture of the Essential Subsystem at the highest level. The notation used in Figure 25 is a variation of that used in (7:175) and is included in Appendix A. The directed arrows between modules indicates compilation dependency where the module at the source of the arrow depends on the module at the destination of the arrow. Of particular interest is the Environment_Types module, which all modules in the Essential Subsystem depend on directly or indirectly with the exception of the Generic Multiple Object Manager. Appendix A presents the entire physical design of the Essential Subsystem in a series of module diagrams.

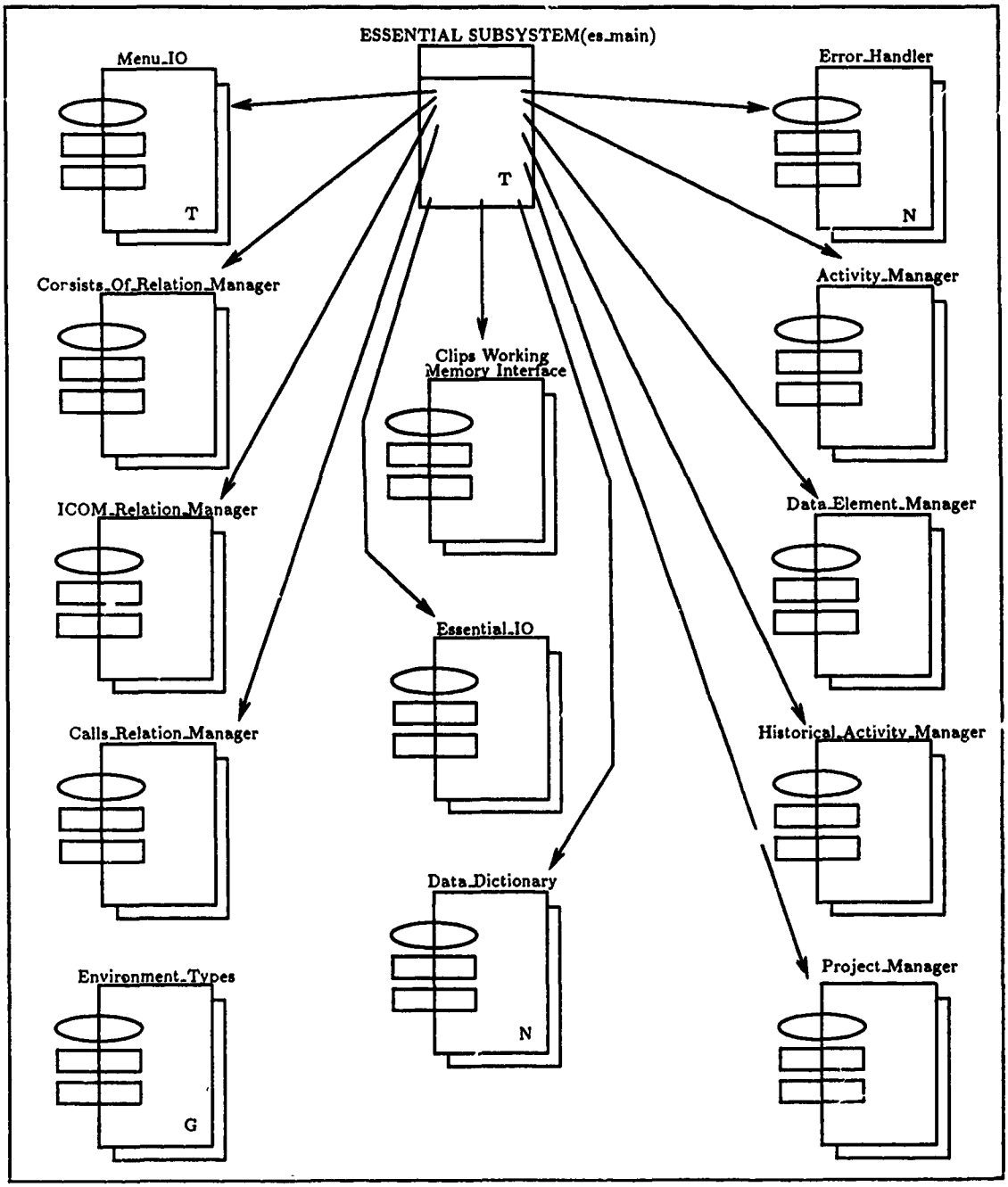


Figure 25. Essential Subsystem Top Level Module Diagram

Environment_Types. The package *Environment_Types* provides global base types, global exception handling variables, one operation, and some instantiations of the Generic Multiple Object Manager. Each of these features has widespread use throughout the Essential Subsystem and SAtool II as well.

Of particular interest are the several instantiations of the Generic Multiple Object Manager. Each of the following manager classes is an instantiation of the Generic Multiple Object Manager.

- Data_Buffer_Package. Instances of this manager class are used for managing multi-line fields of 25 characters (e.g., multiple occurrences of activity names, data element names, etc.).
- Text_Buffer_Package. Instances of this manager class are used for managing multi-line fields of 60 characters. The 'description' and 'changes' attributes of the activity class and data element class respectively are implemented by instances of this manager class.
- Fact_Buffer_Package. As the name implies, instances of this manager class are used to manage essential data model facts destined for either CLIPS Working Memory Interface or Essential IO. Facts extracted from a physical file by Essential IO for restoring the data structures are also managed in this manner.

These manager classes are available by any package or procedure with visibility to Environment_Types. The reasoning for placing these manager classes in the Environment Types package is the same as the reasoning for placing base types in the package. These manager classes have widespread use throughout the SAtool II system. By placing them in a "global" package, multiple occurrences of identical Generic Multiple Object Manager instantiations are avoided and the size of the SAtool II object file is reduced.

The Environment_Types package has compilation dependency on only the Generic Multiple Object Manager and the system Calendar package. Of special note is that all modules/packages in

the Essential Subsystem and SAtool II except the Generic Multiple Object Manager have direct or indirect compilation dependency on this package.

Generic_Multiple_Object_Manager This generic package is a modified version of the software component 'Queue Nonpriority Balking Sequential Unbounded Unmanaged Iterator' presented in (5). Several significant changes are made to the component to fit the requirements:

- The passive iterator supplied with the component is replaced with an active iterator. The source code for the active iterator can be found in (5:158).
- The procedure 'Set_Item' is added to permit an item to be updated in place.
- The procedure 'Add' is modified to include the iterator type as an 'out' parameter which is left pointing to the item just added.
- The procedure 'Remove_Item' now uses an access type in determining the item to delete. It previously used a position number.

The Essential Data Model Object Classes. This discussion includes all the object classes illustrated in Table 2 with the exception of the project class which is modeled as a simple type within Environment_Types. Each of the classes is implemented with an Ada record type, but there is no encapsulation of that type. Thus, each of the following six packages implements an unencapsulated object class.

- Activity_Class
- Data_Element_Class
- Historical_Activity_Class
- Calls_Relation_Class
- Consists_Of_Relation_Class
- ICOM_Relation_Class

The Essential Data Model Object Class Managers. As discussed in the previous chapter, seven different object classes are identified from the essential data model. Because there exists but one instance of the project class in the essential data model and SAtool II, its implementation is different from the others. The package *Project Manager* simply encapsulates one instance of the project class within its body. The project class itself is not included in the encapsulation (i.e., its implementation is not hidden within the body), since it is readily accessible from the Environment_Types package. Therefore, the specification of Project Manager package consists of only two operations:

- Set_Project_Name
- Value_Of_Project_Name

In effect, the above methodology results in the implementation of an *Abstract State Machine.* In this context, an Abstract State Machine is considered to be the encapsulation of a single instance of an object class where state information (i.e., the object) is retained in the package body (6:238). In other words, there is only one object of this type in the entire system that is being modeled.

However, the remaining six manager mechanisms are implemented in a very different manner, because they manage more than one instance of an object class.

1. The Generic Multiple Object Manager package, which is a generic or parameterized class, is instantiated with a particular object class. This results in the creation of an object class – not an object. If only primitive operations were desired on this new object class, the process could be halted here, and the user could simply declare an instance of the object class to obtain a "simple" manager mechanism. However, this is not the case for the objects classes from the essential data model, since more complex composite operations on the contents of the managers are required.

2. The next step then is to declare an instance of the object class and encapsulate it. Since there is no requirement for multiple managers of any one type, the manager itself is placed within the package body. This results in an Abstract State Machine being implemented for each of the six manager mechanisms.

The definition of an Abstract State Machine used in this research does not allow for the exporting of any types (6:238), but due to the unique requirements of SAtool II, this rule is intentionally violated for each of the manager mechanisms. Just as the Generic Multiple Object Manager exports an iterator type that allows direct access to an item, each of the Abstract State Machines exports a pointer type which permits direct access to an item within the Abstract State Machine. Each of the following packages are therefore implemented as Abstract State Machines that export a pointer type:

- Activity_Manager
- Data_Element_Manager
- Historical_Activity_Manager
- Calls_Relation_Manager
- Consists_Of_Relation_Manager
- ICOM_Relation_Manager

Essential_Fact_Utilities. The information that is stored in the manager abstract state machines represents the essential (fundamental) part of an $IDEF_0$ model. As discussed in the last chapter, this information must be extracted from the managers for output to a file for permanent storage or for input to the CLIPS/Ada working memory for syntax checking. The required format for input to the CLIPS working memory is CLIPS facts. Thus, in order to conserve implementation time, CLIPS facts are also chosen as the representation method for the output file that stores the entire essential data model (i.e., the project).

Thus, for each manager abstract state machine, two procedures are required: one procedure to retrieve facts from the manager and another procedure that restores the facts back into the manager mechanism data structures when SAtool II restores a project. All of these procedures for all the managers are located in the Essential_Fact_Utilities package. The procedures are located in a this common package because of their common functionality, and because some of the procedures require visibility to more than one of the managers. The requirement for multiple visibility is the reason that these operations are not part of the semantics (i.e., the methods) of their respective managers. Otherwise, if the operations are part of the object semantics, increased coupling among the managers would result. Consequently, to reduce the coupling among the managers, these procedures are implemented as *free subprograms* or utilities (7:126) and placed in a separate package.

The medium by which facts are transferred into and out of this package is an instance of the Fact_Buffer_Package that is located in the Environment_Types package. Therefore, client programs either get a buffer of facts from a procedure in this package or they pass a buffer of facts to a procedure in this package.

For example, Essential_Fact_Utilities has two procedures related to the ICOM_Relation_Manager abstract state machine.

- Retrieve_ICOM_Facts. This procedure first examines the input parameter'Type Facts Flag'. Based on the flag setting (True or False), the procedure retrieves one of two different sets of facts and inserts them into another parameter, 'Fact Manager', which is an instance of the Fact_Buffer_Package. If the flag is true, only facts for the expert system are inserted in the 'Fact Manager'. If the flag is false, only the facts necessary to permanently store the state of the essential data model (i.e., the $IDEF_0$ model) are inserted into the Fact_Manager. This procedure is invoked by a client program whenever the user saves the project he/she is working on, or when the user wishes to check the syntax of the project (i.e., the current $IDEF_0$ model).

- Restore_ICOM_Facts. This procedure accepts as input a buffer of icom facts representing state information. These facts are retrieved from a file by the 'Restore Project' procedure in the Essential_IO package. These facts are then restored into the ICOM_Relation_Manager by this procedure. This procedure is normally executed as one of a sequence of events in the initialization of SAtool II when a previous project is loaded from disk.

Similar pairs of procedures for each of the manager mechanisms exist within the Essential_Fact_Utilities package. See Appendix A for a list of the procedures not yet implemented in this research.

CLIPS_Working_Memory_Interface. This package provides the interface for the Essential Subsystem to the CLIPS/Ada expert system. It is the only package in SAtool II that has visibility to the CLIPS/Ada expert system operations. This visibility is achieved by the context clause "with Embedded_Clips;" (32:17). This context clause permits the CLIPS Working Memory Interface package to have direct access to the actual working memory of the expert system via the operations found in the specification of the Embedded_Clips package.

The following procedures are included in the CLIPS Working Memory Interface package specification:

- Initialize_Clips. This procedure is an initialization procedure that must be executed prior to any of the other operations associated with CLIPS/Ada.
- Assert_All_Facts. This procedure calls each of the 'retrieve' operations in the Essential_Fact_Utilities package. It controls the retrieval of the facts and their assertion into the CLIPS/Ada working memory.
- Display_All_Facts. This procedure simply displays the contents of working memory.
- Execute_CLIPS. This procedure begins the recognize-act cycle of the CLIPS/Ada forward chaining inference engine.

- Clear_CLIPS. This procedure clears the working memory of CLIPS/Ada.

At this time, only a subset of the essential data model information is loaded into the working memory for syntax checking by the procedure Assert All Facts. Appendix G presents a script file that shows the subset of facts that are loaded.

Essential_IO. This package provides the necessary operations for SAtool II to store essential data model in a file and to load essential data model information from a file into the managers. The following operations are included in the Essential_IO package specification:

- Save_Project. This procedure controls the storing of the $IDEF_0$ model state information to a file. It obtains the state information by calling operations within the Essential Fact Utilities package.
- Restore_Project. This procedure controls the restoring of a project from disk to the Ada data structures. Again, several operations within the Essential Fact Utilities package are used to load the information back into the Ada data structures which represent the managers. Note that the project information, in this case, is only essential data model information. The storage and retrieval of drawing information is to be handled by a separate package (40).

All of the state information from the Project Manager and the ICOM Relation Manager can be retrieved and restored by these operations. In addition, a small subset of the information stored in the Activity Manager is also retrieved and restored. However, no state information from the other five managers is currently handled.

Appendix F presents the output file format for a sample $IDEF_0$ model. The state information in this file is represented as CLIPS/Ada facts and illustrates the subset of state information facts that the Essential Subsystem currently captures.

Data_Dictionary. This package provides operations to create an ASCII text representation of either an activity data dictionary entry or a data element data dictionary entry. Operations are also provided to output all data dictionary entries to file for both activities and data elements. As stated in the last chapter, this output can be used as an alternative means for storing the state information. However, procedures to restore the state information from the data dictionary entries would then have to be developed and added to the Essential_IO package.

Unfortunately, time constraints prohibited the implementation of this object. However, Appendix A presents some ideas on how to possibly implement this object for future research efforts.

Error_Handler. The Error Handler object is the single focal point within SAtool II for error handling. There is one error handler object for the entire SAtool II system. However, because integration with SAtool II is not accomplished, a separate version of the error handler object is to be created for use within the Essential Subsystem alone. Upon integration with SAtool II, the error handler from this research and (40) are to be merged into a single object.

The error handling methodology used is the same for all packages in SAtool II. There are four global variables within the Environment_Types package: 'error number', 'error location', 'known exception occurred flag', and 'unknown exception occurred flag'. In addition, each package in SAtool II also has a corresponding exception naming that package within Environment_Types. Whenever an exception occurs in any procedure of any object, the procedure updates those variables to reflect the type of exception that has occurred and then, it raises the package exception in the Environment_Types package. This exception then propagates to the SAtool II main program which invokes the Error_Handler object. Its function is to examine the contents of the global variables, inform the user of the problem, and then take the appropriate action based on the user response.

Due to time constraints, this package is not implemented. However, additional information on the package implementation can be found in Appendix A.

Expert System

The implementation of the expert system is highly dependent on the implementation of the Essential_Fact_Utilities package, because the package contains the operations necessary to convert the $IDEF_0$ model state information into CLIPS/Ada facts. As explained in Chapter Four, the facts destined for the working memory of the expert system may have different information than those facts that represent the state information.

This difference is illustrated by comparing the file presented in Appendix F, which contains facts for state representation, to the script file in Appendix G which includes a display of the syntactical facts of nearly identical $IDEF_0$ model.

The following list illustrates those syntactic facts of an $IDEF_0$ model that the Essential Subsystem can currently retrieve:

1. The activity name of each activity in the model.
2. The number of input, output, control, and mechanism arrows for each activity.
3. The project name.
4. The description and activity number for each activity.
5. The activity hierarchy in terms of parent - child relationships.
6. The relationships between every activity and every data element in the model.

The knowledge (rule) base of the expert system is presented in Appendix D. Appendix G presents a script file that illustrates the syntactical checking of the $IDEF_0$ model shown in Appendix F, except that the version checked in Appendix G has been slightly modified to introduce a few errors. The following list illustrates the syntactic features of an $IDEF_0$ model that the Essential Subsystem can check via the rule base that is depicted in Appendix D:

1. The number of input, output, control, and mechanism arrows for each activity are checked to insure a valid number of them are in place.
2. The description and activity number of an activity are checked for their presence.
3. The project name is checked for its presence.

Obviously, the implementation of this small subset of rules is rather simple. However, time constraints prohibited expanding the rule base any further within the available research time. However, the feasibility is adequately demonstrated.

Documentation Standards

The documentation standards in the *System Development Documentation Guidelines and Standards Draft # 4* (16) are intended for doc: menting a system based on a functional design approach. Standards are provided for file headers and for Module/Subroutine/Procedure headers (16:32–35). Although the file header format is adequate, the Module/Subroutine/Procedure header standard is not sufficient for an Ada based object oriented design. Specifically, there is no standard for documenting objects which, in Ada, are commonly implemented by a package. In addition, the Module/Subroutine/Procedure header standard does not include a requirement for including the its order-of. Therefore, Appendix E presents a modified version of the Module/Subroutine/Procedure header format and a new header format for use with packages. Both formats are used to document the source code presented in Volume II of this research. Furthermore, in order to increase the maintainability of the source code, the Ada *USE* clause is strictly avoided. Thus, all subprogram calls are explicit in their package origin.

Order Of Analysis

The term *order of* in this context refers to the order of magnitude in terms of time complexity for a given program. For the worst case or 'upper bound' time complexity, the Big-O notation is used (26).

Big-O Definition:
A function $f(n)$ is of order $O(g(n))$ if and only if there exist constants, $c > 0$ and $n_0 \geq 0$, such that

$$f(n) \leq c \cdot g(n) \qquad \forall n \geq n_0$$

$f(n) = O(g(n))$ says that $g(n)$, multiplied by some constant, c, gives an upper bound on $f(n)$. (26)

A "complete" order-of analysis requires that each statement or block of statements of a program be approached separately for analysis. The order of for the entire program is then determined by combining the results of earlier analyses to obtain the overall Big-O (1:21–26). Any individual subprograms contained in the program are decomposed into smaller blocks of code to simplify their order of analysis. This process is categorized as an "inside-out" approach to program analysis. The 'program' of concern in this case is the Essential Subsystem Test and Demonstration Program which is primarily a menu program to model the SAtool II graphical user interface. However, this program is developed solely for testing and demonstration purposes and will not be part of the final SAtool II tool. Therefore, the order of analysis is performed only on those subprograms that will eventually be part of SAtool II. The results of the analysis are embedded as comments within each of the subprogram headers as well as the subprograms themselves.

Table 4 provides a sample of the time complexity of some of the operations within the Essential Subsystem. In this case, for each package, the operation with the greatest time complexity is illustrated. Where there are multiple operations with the same apparent maximum time complexity in a package, an arbitrary choice is made. Of special note is that in the worst case only cubic time

Table 4. Greatest Time Complexity Per Package

Package.Operation	Time Complexity
Activity_Manager.Create_Activity	$O(a * z)$
Data_Element_Manager.Create_Data_Element	$O(d)$
Historical_Activity_Manager.Create_Historical_Activity	$O(h)$
Calls_Relation_Manager.Create_Calls_Relation_Tuple	$O(c)$
Consists_Of_Relation_Manager.Value_Of_Consists_Of_Id	$O(s * s * k)$
ICOM_Relation_Manager.Create_ICOM_Relation_Tuple	$O(i)$
Project_Manager.Set_Project_Name	$O(1)$
Essential_Fact_Utilities.Restore_Activity_Facts	$O(a * max(x, z * (a * z)))$
Essential_IO.Restore_Project	$O(max((i * i), a * max(x, z * (a * z))))$

Table 5. Order-Of Variables

Variable	Meaning
a	The number of activities in the Activity Manager
d	The number of data elements in the Data Element Manager
h	The number of Historical Activities in the Historical Activity Manager
c	The number of Calls Relation Tuples in the Calls Relation Manager
s	The number of Consists Of Relation Tuples in the Consists Of Relation Manager
k	The number of components of a data element
i	The number of ICOM Relation Tuples in the ICOM Relation Manager
x	The number of lines in an activity's description
z	The number of children activities that an activity possesses

is exhibited. Table 5 defines the variables used in Table 4. Table 6 presents a summary of the order-of results for all of the 117 operations of the Essential Subsystem.

Note that the Clips Working Memory Interface package is omitted from Table 4, because its procedures make direct calls to CLIPS/Ada operations whose analysis is not part of this research. Thus, the order-of for each of the operations in the package is unknown. This also accounts for five of the six "unknown" operations in Table 6. The other operation of unknown time complexity is the 'Get Date Time Stamp' operation contained in the Environment Types package which calls an operation within the system dependent Calendar package.

Table 6. Greatest Time Complexity Per Package

Time Complexity	Number of Operations
Cubic Time	2
Quadratic Time	11
Linear Time	37
Constant Time	61
Unknown	6

Testing

A bottom-up, incremental approach (39:410–413) to testing is used in this research. This approach is used for two reasons. First, the most critical objects implemented are the manager objects which hold the $IDEF_0$ model state information. These objects are at the lowest level in terms of visibility within the physical design. Since the lowest level objects are implemented first, a bottom-up approach to testing is considered most applicable. Second, a top-down approach is not feasible in this case since the "top" of SAtool II is already under development in other research (40).

Testing for the Essential Subsystem begins immediately after the first unit (subprogram) is implemented and continues past the implementation of the last module. Thus, the testing phase significantly overlaps the implementation phase of the Essential Subsystem.

The bottom-up testing methodology is followed by constructing a driver program (es_main) that provides the necessary input to the modules and units to be tested. Once es_main is constructed, a module is chosen for implementation, and a corresponding menu of its operations is added to es_main. Each of these operations is stubbed out at the beginning. As each unit of the module is implemented, its stub is replaced by the actual operation and is tested.

The incremental approach is used once all the operations of the first module are implemented. Instead of creating another driver program for the second module, the same driver program (es_main) is used by adding another menu of operations. In this manner, an incremental build of the Essential Subsystem occurs as each subsequent module is implemented. If the module

just added communicates with (i.e., has any relationships with) any of the other modules already added, these relationships are tested. Since the interface to the expert system is modeled by the CLIPS Working Memory Interface package/module, the testing of the expert system occurs when the relationships of the CLIPS Working Memory object are tested.

Once all the modules (objects) are implemented and tested, the test program (es_main) doubles as both a demonstration program and a validation program.

Due to the time constraints imposed on this research, the testing that is performed is primarily functional in nature. Some limited boundary testing is performed, especially for the menu input routines. However, formal testing in terms of statement coverage, codepath coverage, etc. is not performed. In addition, a formal proof of correctness is considered outside the scope of this research. Instead, sample $IDEF_0$ models are created and loaded into the Essential Subsystem. Several of these models are created and manipulated by the program to validate that no unexpected results occur. Appendix G presents one of the functional test cases and the results of the test.

Integration with SAtool II

Had additional time been available for this research and the concurrent research (40), the integration of the Essential Subsystem with the remainder of SAtool II could have proceeded. Instead, the Essential Subsystem Test and Demonstration Program is developed which models some of the functionality of SAtool II. Some of the available functions can be seen by examining the script file presented in Appendix G. However, a full appreciation of the multitude of available operations can not be achieved without examining the source code in Volume II of this research.

The outermost menu of choices for the Essential Subsystem Test and Demonstration Program attempts to model some of the "mouse selections" that would be available through the graphical user interface of SAtool II. The GUI for SAtool II is illustrated in Figure 26 which depicts the overall architecture of SAtool II. The program menu choices "Add_Box" and "Connect_Two_Boxes"

implement operations that are considered part of the layer below the GUI – "The Inter-Model and Intra-Model Macro Operations and Project Integrity Constraint Management" part of SAtool II. It is this part of SAtool II that enforces the integrity constraints that are discussed in the Relationships and Visibilities section in Chapter Four.

The series of inner menus in the Essential Subsystem Test and Demonstration Program permits the user direct access to the Essential Model section and the File and Expert System Utilities section of the SAtool II architecture shown in Figure 26. Of course, once full integration is achieved, such access should definitely be prohibited.

Summary

This chapter presents the implementation and testing of the Essential Subsystem which includes an Ada based expert system.

The implementation of each of the objects in the Essential Subsystem is discussed. A module diagram is used to illustrate the highest level in the physical design of the Essential Subsystem. The implementation of the object class managers as Abstract State Machines is also discussed.

The implementation of the expert system is also discussed. Only a limited number of $IDEF_0$ syntactic features are checked, because time constraints prohibited the development of all the necessary fact retrieval operations.

The documentation standards are augmented with a revised subroutine header and a new package header format. Both formats are used throughout this research and are presented in Appendix E.

The order of for each and every subprogram in the Essential Subsystem is performed with the exception of those subprograms implemented for testing and demonstration purposes only and the subprograms in the Clips Working Memory Interface package. A partial order of analysis is performed for these operations because the order of for the CLIPS/Ada operations is not provided in

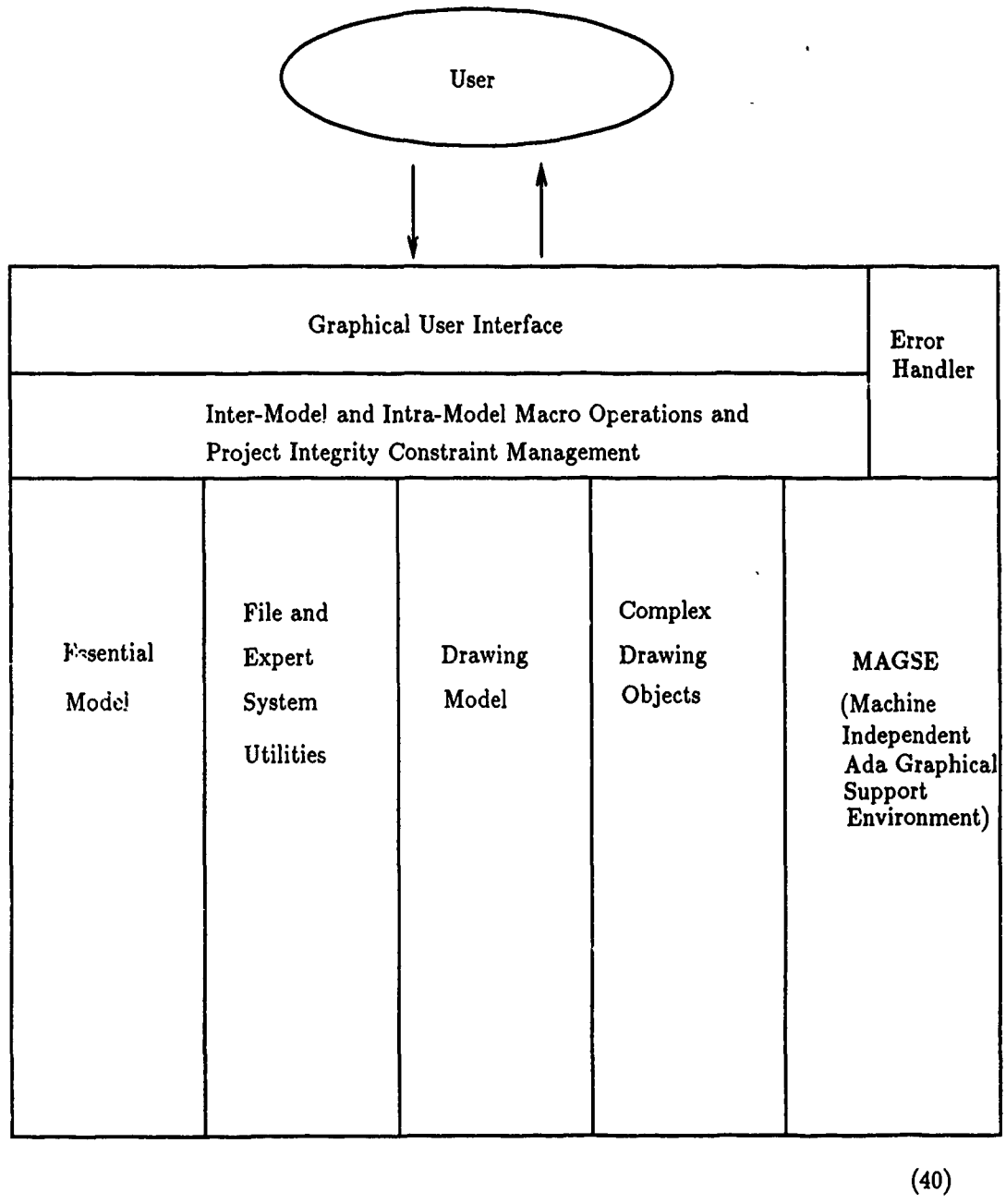


Figure 26. SAtool II Overall Architecture

the documentation. Time constraints prohibited performing an order of analysis for the CLIPS/Ada operations. The order of analysis for all the operations is documented using the Big-O notation (26).

Testing is performed in concert with the implementation. As each unit is created, it is added to a module within a test program (es_main). When all the units of a module are completed, the first unit of the next module is then implemented. Any relationships that exist between modules are tested as the units of the module are added. Thus, an incremental build of the Essential Subsystem is accomplished using a bottom-up approach.

Because time constraints prohibited the full implementation of the Essential Subsystem and thus integration with the rest of SAtool II (40), the Essential Subsystem Test and Demonstration Program was implemented to demonstrate some of the proposed functionality of SAtool II.

VI. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

Introduction

This chapter first presents a summary of this research investigation. Conclusions about the research and recommendations for further work are also included.

Research Summary

This investigation resulted in several accomplishments in relation to the design and implementation of the Essential Subsystem for SAtool II:

- A revised $IDEF_0$ Essential Data Model was developed. This included a revised $IDEF_0$ Activity Essential Data Model and a revised $IDEF_0$ Data Element Essential Data Model.
- A revised AFIT Data Dictionary Format was developed. This included revised data dictionary entry formats for both an activity and a data element.
- An object oriented design of the Essential Subsystem was developed which demonstrated the following:
  - An entity relationship diagram can be used as the basis for an object oriented design.
  - Relationships in an entity relationship diagram can be modeled as objects in an object oriented design.
  - The design of an expert system can be integrated with the object oriented design process by modeling the interface to the expert system as an object.
- An implementation of the Essential Subsystem design called the Essential Subsystem Test and Demonstration Program was developed which demonstrated the following:
  - The feasibility of using of Ada and object oriented design techniques in the implementation of a CASE is possible.

- The feasibility of using Ada for an expert system is possible.
- The feasibility of representing the state of an $IDEF_0$ model as CLIPS facts, and the feasibility of representing the syntactical features of an $IDEF_0$ model as CLIPS facts are both possible.

In the process of making the above accomplishments, several critical design and implementation decisions were necessary:

- The most critical decision is the viewpoint from which the problem is observed. The outside view of an $IDEF_0$ can result in one object with many component objects. The view chosen for this research is an inside view which results in the many objects that are part of the design.
- The decision to select the inside view permits the retrieval of state information to be much simpler, since the information is not embedded within the structure of a single object as it is for the outside view.
- The joint design decision (40) not to permit visibility between the manager objects within the drawing model (SAtool II) and the manager objects within the essential model (Essential Subsystem) is significant. Furthermore, no manager within the drawing model has visibility to any manager in the essential model and vice versa. This decision eliminates coupling among the manager objects but increases the complexity of the client program that must manipulate the objects, because the responsibility for maintaining the integrity constraints among the objects is exported to the client program. In this case, the client is the layer just below the Graphical User Interface layer of SAtool II shown in Figure 26.
- The decision not to encapsulate the essential data model object classes permits a layer of abstraction to be removed.

- The decision to allow direct access to the items stored in the manager mechanisms is a violation of the principles of information hiding and encapsulation. However, it reduces the time complexity of some operations to O(1) time.
- The decision to model some 'many to many' relationships in an entity relationship diagram as objects permits the manager mechanisms to be implemented with no coupling among themselves.
- The decision to store the state information of an $IDEF_0$ model as CLIPS/Ada facts is made primarily due to time constraints.

Conclusions

The review of both the abstract data model of the $IDEF_0$ language and the associated data dictionary formats prove to be worthwhile, since several inconsistencies and inadequacies are identified and corrected. Without identifying and correcting these deficiencies early in the research, serious design and implementation problems would have resulted.

The partitioning of the essential model information from the drawing model information continues through the design and implementation phase. This shows that the partitioning of the $IDEF_0$ language by the $IDEF_0$ Abstract Data Model does, in fact, correctly model the separation of the essential (fundamental) information of an $IDEF_0$ model from the drawing information of an $IDEF_0$ model.

An entity relationships diagram can be used as the basis for going from the requirements phase to the design phase of the software development life cycle. However, the technique devised in this research for mapping E-R constructs to objects in an OOD can not be applied to any ERD. The methodology only includes techniques for those E-R constructs that appear in the abstract data model of the $IDEF_0$ language. Even if similar E-R constructs appear in another E-R model, it is not clear whether this technique can be applied.

The modeling of relationships as objects reduces coupling among objects. Coupling can be eliminated entirely if the responsibility for maintaining the integrity constraints among related objects is exported to the client program. If that responsibility is not exported, coupling is reduced to a lesser degree.

The Generic Multiple Object Manager is used in the creation of several abstract state machines that violate the principles of encapsulation and information hiding. These violations are the result of the unique requirements of a CASE tool and not inherent to the generic manager itself. In other words, components that do not violate encapsulation and information hiding can be developed with the generic manger, but the generic manager also permits those rules to be violated if the user so desires. Its flexibility is its greatest characteristic.

Both this research and concurrent research (40) demonstrate that Ada can be used in the development of a CASE tool. The use of an object oriented design approach to CASE tool development is also shown to be successful.

The feasibility of Ada in expert system development is clearly evident, since CLIPS/Ada is readily available as a public product as well as being available to the government for free. Its use as an integral component of a CASE tool is demonstrated by the CLIPS Working Memory Interface object which implements an interface between the Essential Subsystem and CLIPS/Ada.

This research shows that the entire hierarchy of an $IDEF_0$ model can be represented by a single implementation of an object oriented design. An earlier version (19) of SAtool only permits a single diagram to be modeled in a single session. Thus, multiple sessions are required to create an entire $IDEF_0$ model. A later version (38) appears to incorporate the necessary activity and data element hierarchy but does not correctly model the multiple decomposition of data elements nor the multiple source-destination relationships among activities and data elements.

During this research, a methodology for performing order-of analysis as well as copious commenting of the source code was used as source code was developed. Although the additional docu-

mentation did impact on the amount of source code produced, the resulting documentation, which includes the source code itself, will be much more maintainable and understandable in follow-on research efforts.

Recommendations

Unfortunately, this research effort did not conclude with an integration with SAtool II that is under concurrent development (40). This is strictly due to the time constraints that are naturally imposed on most research efforts. Therefore, the most obvious recommendation is to perform the integration between the Essential Subsystem and SAtool II.

The objects derived directly from the Essential Data Model are fully implemented with the exception the minor changes required to implement the Error Handler object. Some of the other objects that are part of the Essential Subsystem are not fully implemented and others are not implemented at all. Again, this is simply due to time constraints. Appendix A describes the remaining work that should be accomplished to complete objects in the Essential Subsystem.

The Essential_IO restore operation, as well as the all the Essential_Fact_Utilities restore operations, require that the some of the fields of a fact be in a specific position within the fact string. In other words, if the fields are not in the correct columns, the facts may be rejected. To eliminate the dependency on columns and make the operations more flexible, these operations should be modified to parse the strings for the individual fields of data. Even greater flexibility in fact representation can be realized in the aforementioned routines are modified to accept facts in any sequence instead of the rigid sequence that is currently enforced. However, this flexibility could result in performance drawbacks, since each and every fact would have to be checked to determine its type.

Now that both the essential data model and drawing data model state information are in a "commercial" file standard (i.e., CLIPS facts), applications can be more readily developed for them.

For example, CLIPS rule based production systems could be written to analyze and manipulate the $IDEF_0$ facts. In other words, a CLIPS rule base could be developed to modify the $IDEF_0$ model information represented by CLIPS facts. Then, executing the rule base would modify the facts, creating a new model. Those facts could then be restored back to the Ada data structures. However, this option only becomes feasible if a feature for automatic $IDEF_0$ diagram generation from essential data model information is implemented, since the $IDEF_0$ drawing model information would no longer be compatible with the modified $IDEF_0$ essential model information.

The size of CLIPS/Ada may be of concern when integration with SAtool II is performed. The current size of the Essential Subsystem Test and Demonstration Program is over 1.3 megabytes of which CLIPS/Ada appears to be a significant part. Therefore, either the size of the CLIPS/Ada expert system should be reduced by the means discussed in (32:50–52), or another Ada based expert system such as the one presented in (4) should perhaps be employed.

There are several recommendations for the expert system portion of the Essential Subsystem:

- Complete the knowledge/rule base of the expert system to check all the syntactic features of an $IDEF_0$ model.
- Expand the rule base to include rules to check drawing information. For example, the position of the boxes on the diagram could be checked to see if "dominance" (36:20) is implemented.
- Experts use certain heuristics in determining whether the specific decomposition of a system is "good" or "bad" which is unrelated to the syntactical correctness of the model. For example, by visualizing the $IDEF_0$ model as a tree structure, the height of the various branches of the tree could be examined. In other words, the "balance" or lack of balance in the $IDEF_0$ hierarchy could be one heuristic in identifying areas of concern.
- Expand the knowledge base of the expert system to include knowledge about the specific application being developed. Are there rules or certain syntactical features that are unique

to certain types of systems? Again, future research into this area is the only way to answer the question.

- Can a computer successfully recreate one or more of the drawing models with only the essential data model information, or is there some key construct missing from the essential data model? One way to answer this question is to attempt to develop an algorithm for converting the information in the essential data model into drawing model constructs. If such an algorithm can be developed, it will demonstrate that the essential model does, in fact, contain sufficient information to recreate one or more physical $IDEF_0$ diagrams that represent the model.

There should be a methodology developed to automatically update the version number and date of an activity or data element whenever either is modified. The responsibility for performing the updates can be exported to the client program, or it can be built into the Activity_Manager and Data_Element_Manager.

CLIPS stores all numbers as real numbers in working memory which can be seen in Appendix G. This representation currently poses no problems, because only a couple of rules are written that match with the real numbers. Additional rules will have to written eventually, however. Thus, this potential problem should be examined as soon as possible by developing additional rules that match on the number fields of the $IDEF_0$ facts.

The concept of data elements having subtypes is stated but not shown in (30:3-7). However, the use of subtypes is explicitly shown in (15:17–26). As currently implemented, the Consists Of Manager considers a decomposition to be its subcomponents, but it has no way of determining that it has been supplied with subcomponents or subtypes. The explicit modeling of subtypes would resolve this problem.

As stated at the conclusion of Chapter Three, it was not the purpose of this research to fully analyze and correct the $IDEF_0$ Essential Data Model and the corresponding data dictionary formats. Obvious inconsistencies and inadequacies were identified and corrected. However, it is suggested that additional research be performed to determine if the models resulting from these changes are, in fact, free of any further inconsistencies or inadequacies in modeling the $IDEF_0$ language.

Appendix A. *ESSENTIAL SUBSYSTEM PHYSICAL DESIGN*

This appendix presents the physical design of the Essential Subsystem. The notation used to present the design is a variation of that used for the module diagrams presented in (7:175) and is shown in Figure 27. Since the design presented here is not fully implemented, the status of the individual modules is also discussed.

The series of module diagrams that follow depict the physical structure of the Essential Subsystem. Some of these modules are implemented strictly for testing and demonstration purposes only and will be replaced by operations contained in the Satool II GUI under concurrent development (40). These modules are clearly marked by a 'T' embedded within the module itself. For example, the Essential Subsystem module (es_main) is primarily a menu program.

Any module that has not been implemented is clearly marked with an 'N' as indicated in the module diagram notation. Time constraints prohibited their implementation. These modules are as follows:

- Error_Handler. Note that the implementation for each package will have to be modified in order for this package to be effective. In other words, the 'exception' part of the operations contained in each package must update the global error handling variables contained in the Environment_Types package. If another means for handling exceptions is desired, the Error_Handler object can be omitted, but some error handling methodology should be implemented.
- Data_Dictionary. The implementation for this object can be handled in at least two different ways. It can be implemented as an encapsulated object class whose instance is a single data dictionary entry. It can also be implemented as a group of utility operations that can create both a single data dictionary entry or create a file of multiple data dictionary entries. This file could then be used as an alternative method for representing the state information of the Essential Subsystem if desired.

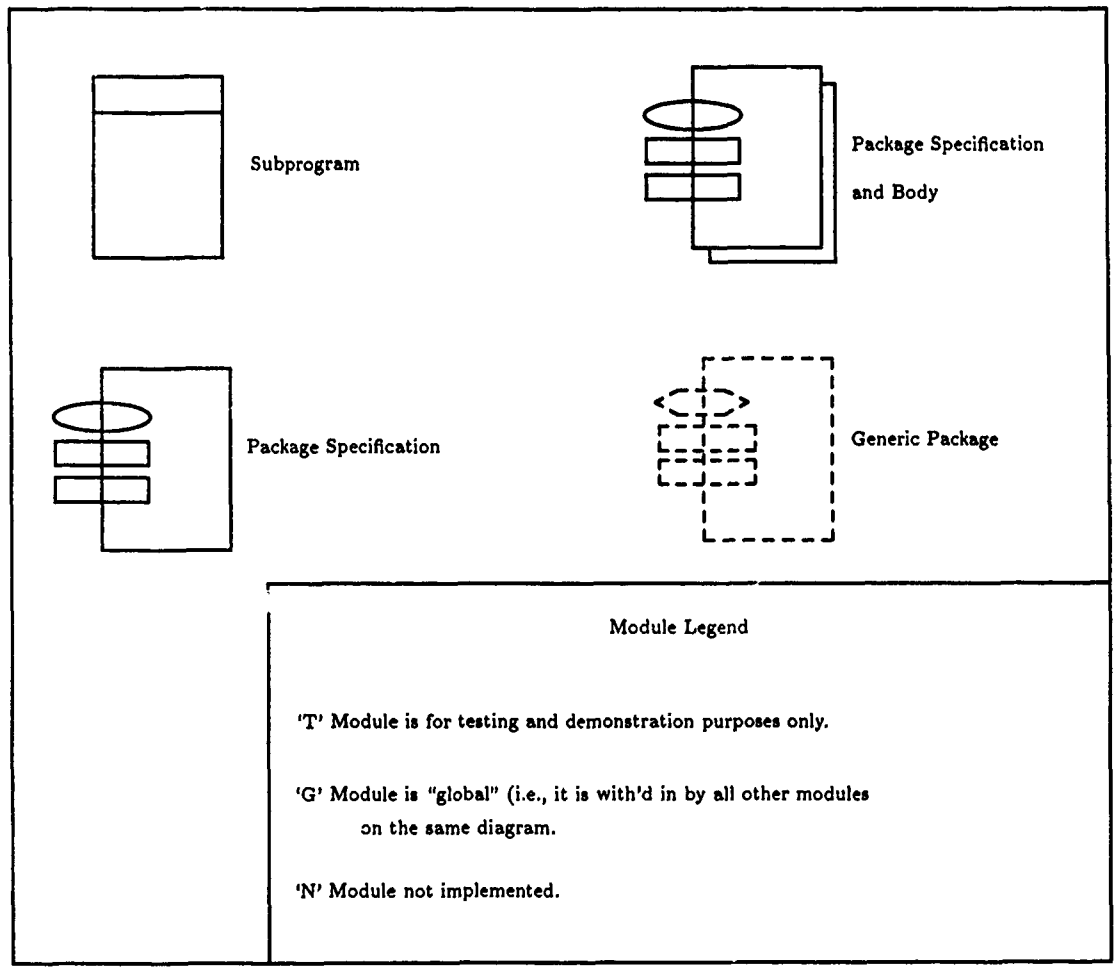


Figure 27. Module Diagram Notation

Some of the modules depicted in the physical design are not fully implemented. The following modules require additional work:

- Essential_Fact_Utilities. The operations 'Retrieve Activity Manager Facts' and 'Restore Activity Manager Facts' currently only retrieve and restore the same small subset of Activity Manager information. The activity name, number, description, and parent-child relationships are the only facts retrieved and restored. The remaining facts contained in the Activity Manager need to be included as well. Retrieving and restoring operations need to be developed for the following managers:

  - Data_Element_Manager
  - Consists_Of_Relation_Manager
  - Historical_Activity_Manager
  - Calls_Relation_Manager

  Operations for retrieving and restoring the information contained in the ICOM_Relation_Manager and the Project_Manager have been fully implemented.

- Clips_Working_Memory_Interface. No additional operations are required. However, the operation Assert_All_Facts needs to be modified to retrieve all the facts from all the managers. This can only be done once the Essential_Fact_Utilities package is completed.

- Essential_IO. No additional operations are needed. However, both of its only operations, 'Save Project' and 'Restore Project', must be modified once all the Essential_Fact_Utilities operations are completed. Currently, the operations are limited to saving and restoring a small subset of the necessary facts to capture all the essential information in an $IDEF_0$ model. Like the Clips_Working_Memory_Interface package, the operations here can only be completed once all the fact utility operations are complete.

In order to present a complete picture of the physical design of the Essential Subsystem, the top level module diagram is repeated here. Note that the compilation dependencies among the objects (other than those associated with es_main) are not shown in Figure 28. However, all subsequent module diagrams depict all the dependencies among all the modules in the diagram.

Of special note is the implementation for the Project Manager depicted in Figure 39. Its implementation differs from the other managers. The absence of a 'Project Class' module is due to the fact that the project class is simply a type with a single field and is contained within the Environment_Types package. Also, the Essential Subsystem, at this time, does not permit more than one project at a time. Thus, there is also no requirement for the Generic Multiple Object Manager. This diagram also depicts the implementation of the Environment_Types package.

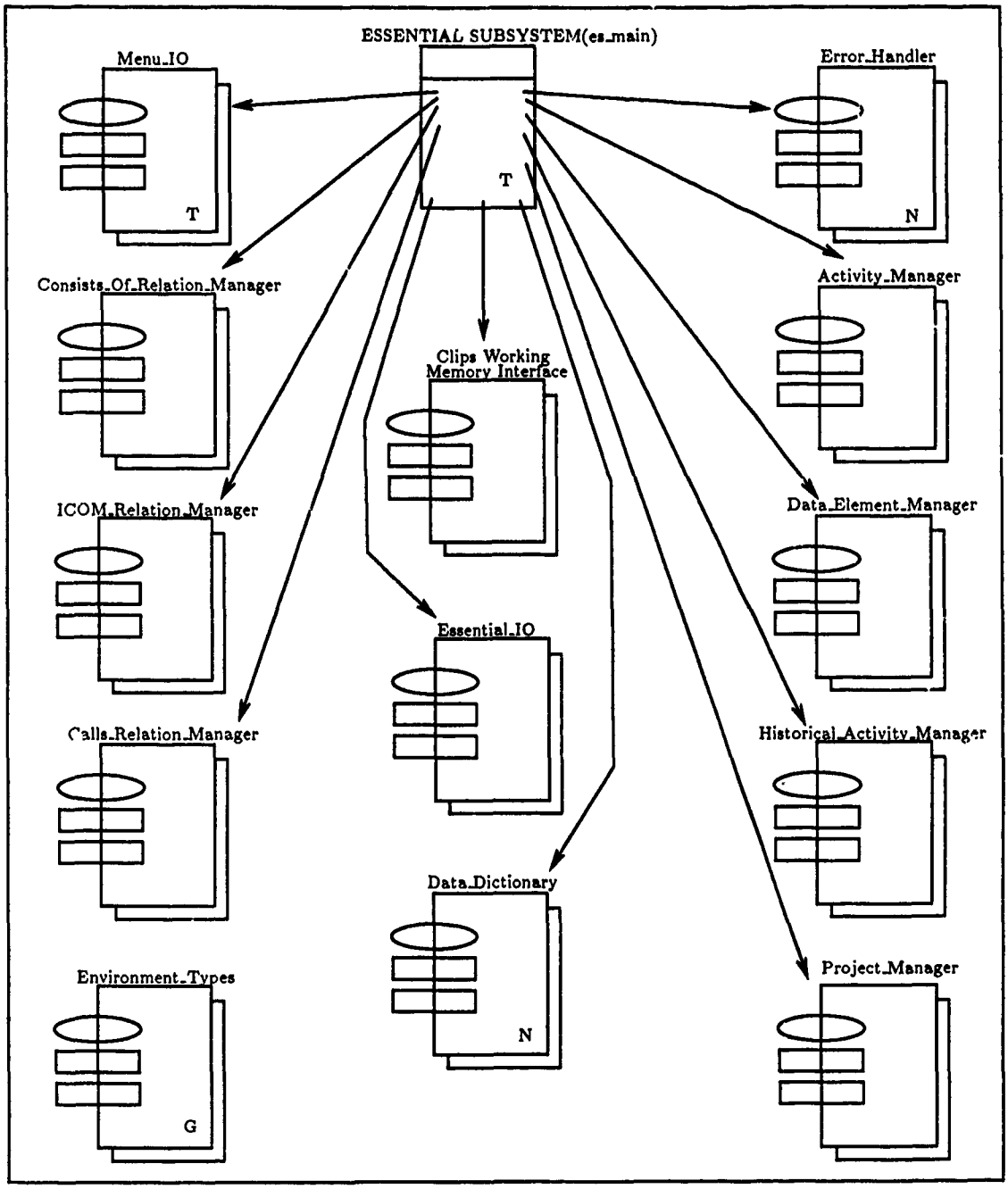


Figure 28. Essential Subsystem Top Level Module Diagram

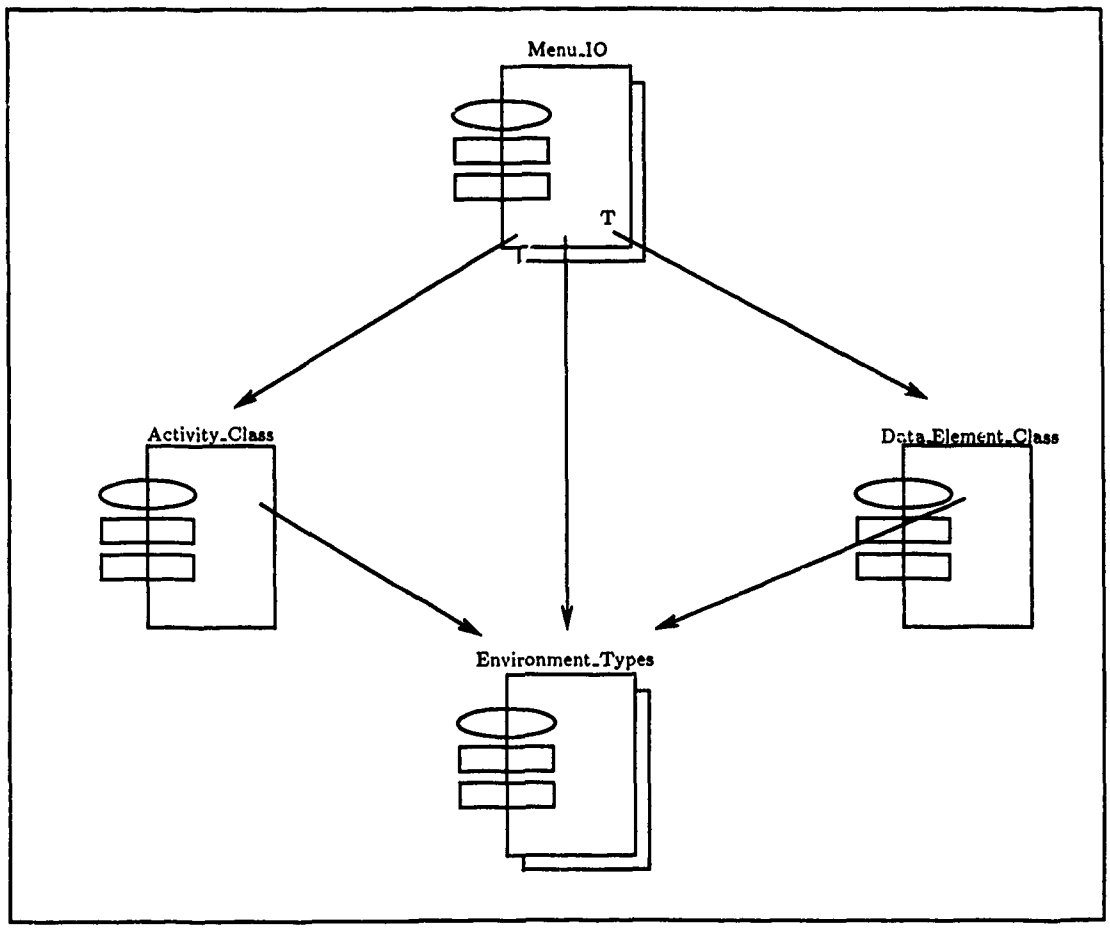


Figure 29. Module Diagram for Menu_IO

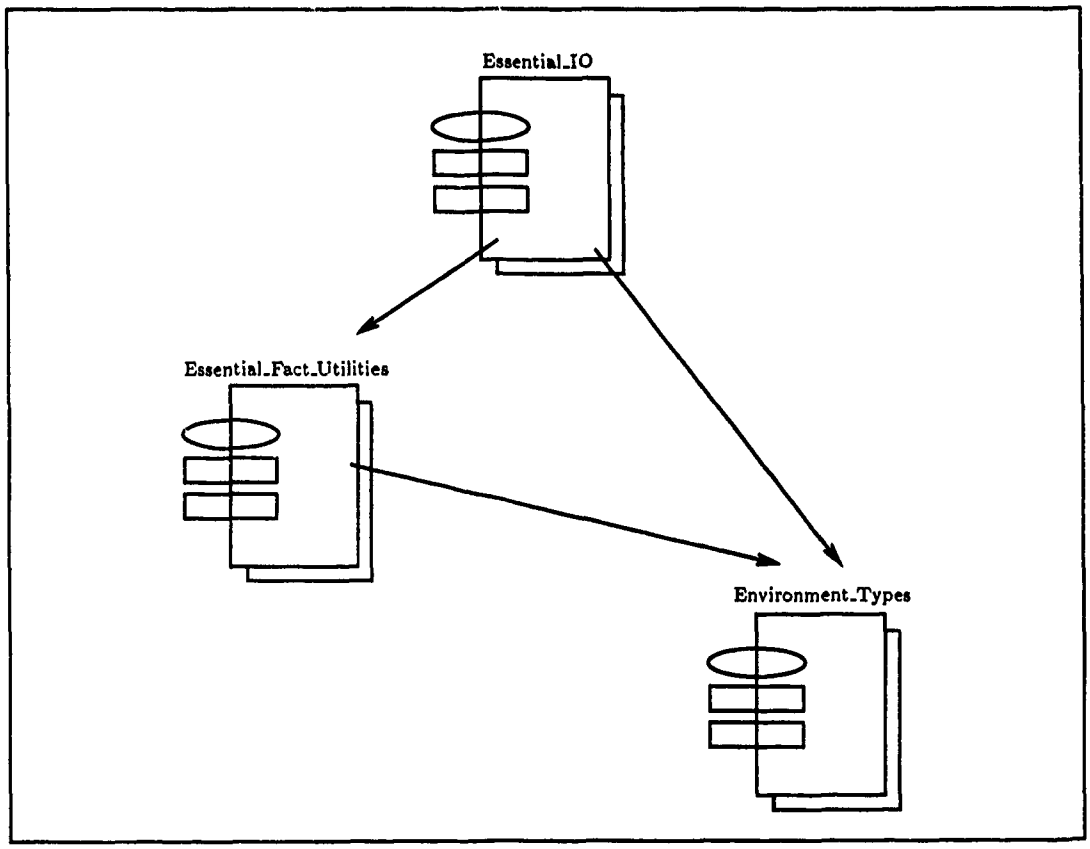


Figure 30. Module Diagram for Essential_IO

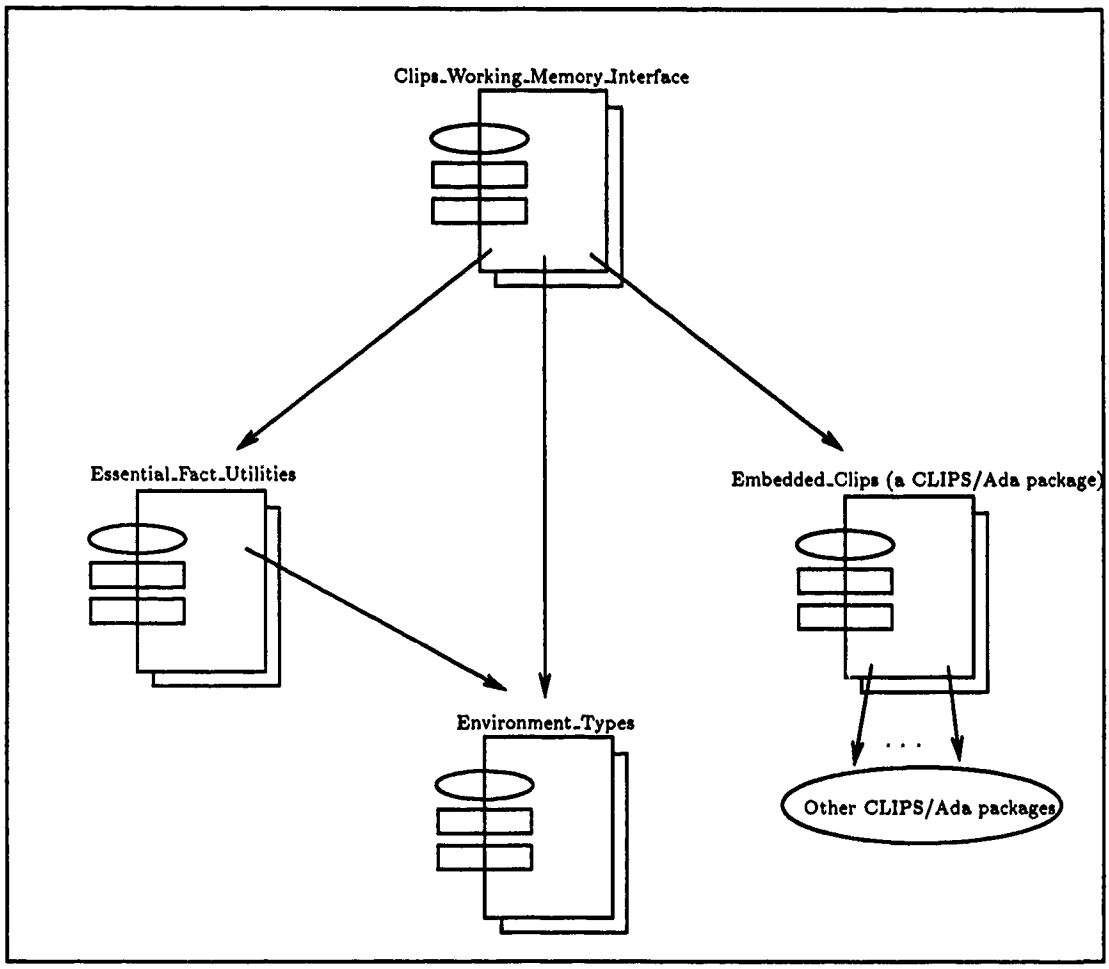


Figure 31. Module Diagram for Clips_Working_Memory_Interface

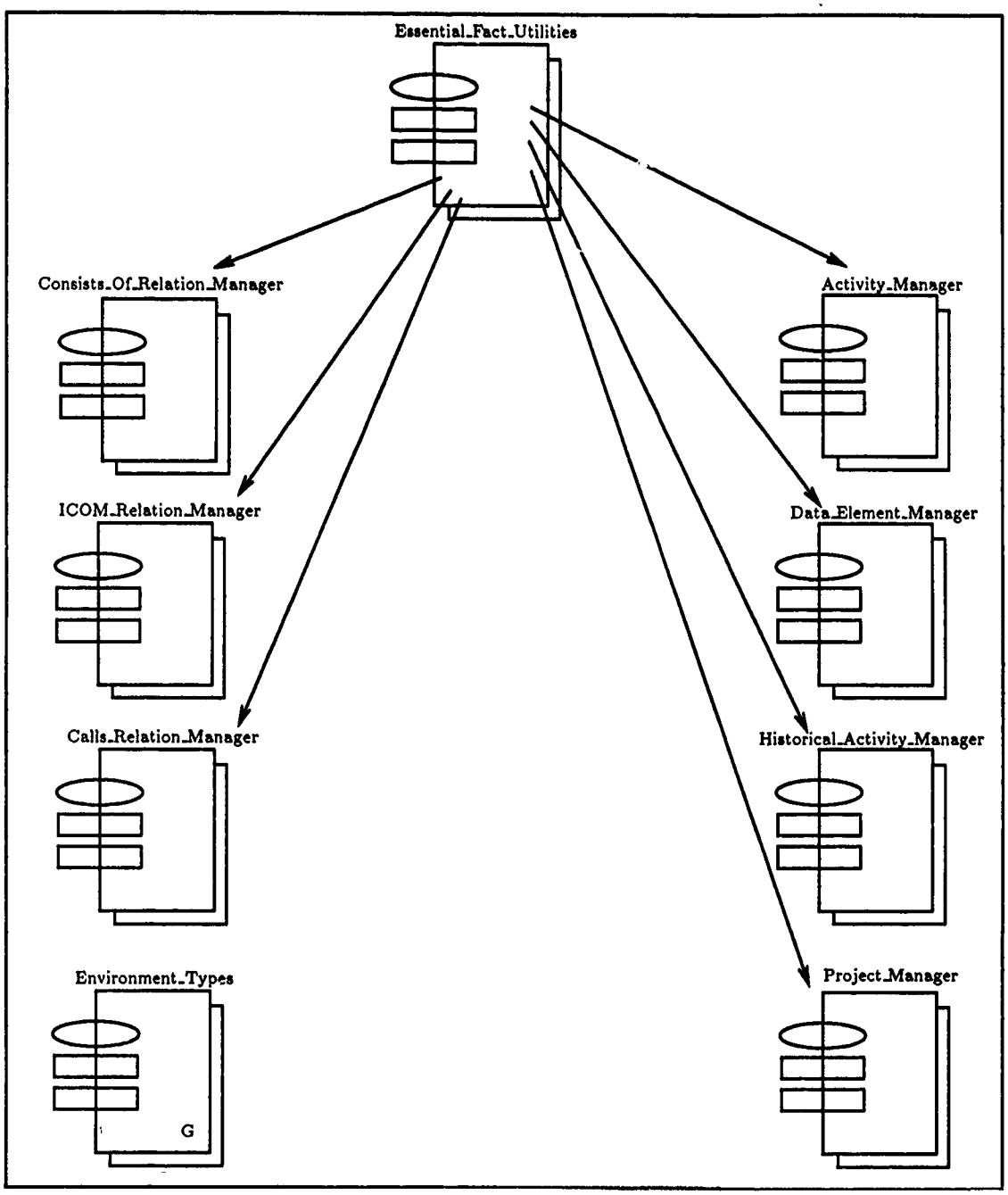


Figure 32. Module Diagram for Essential_Fact_Utilities

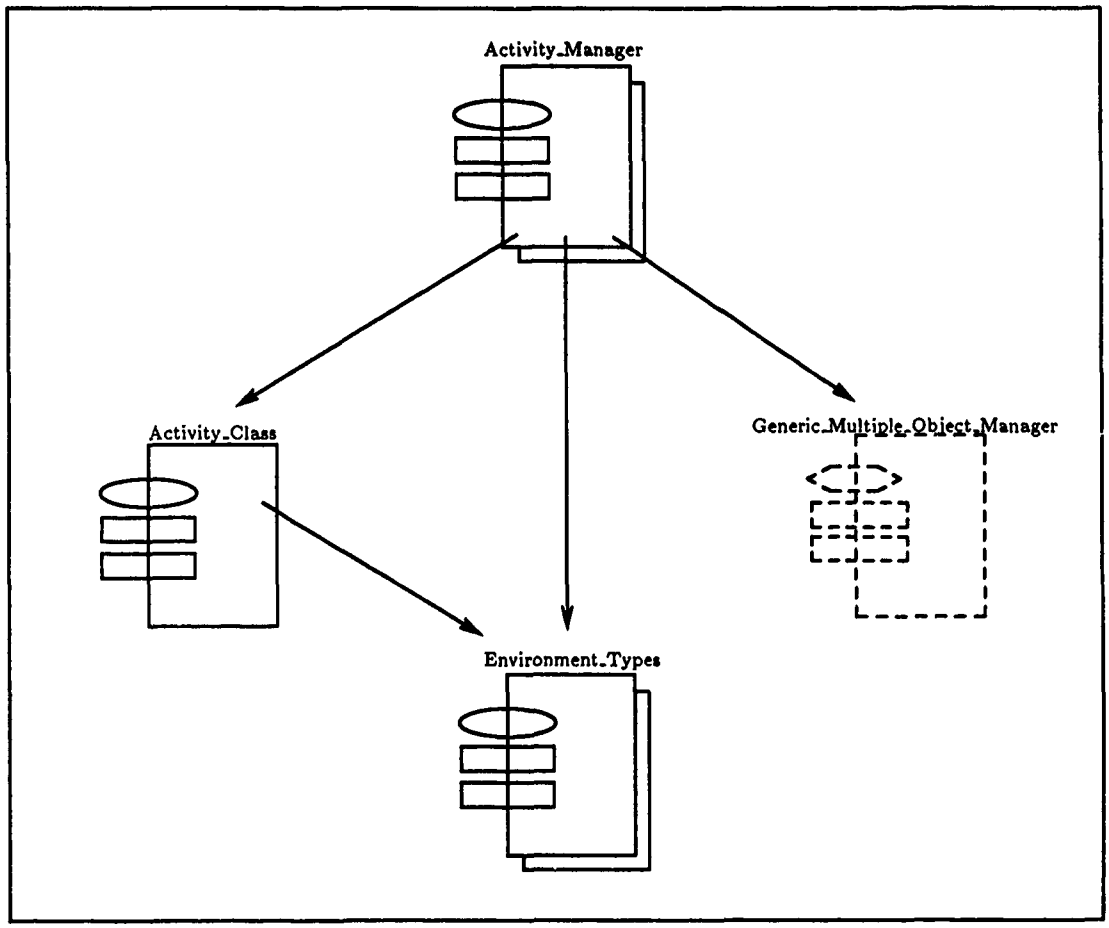


Figure 33. Module Diagram for Activity_Manager

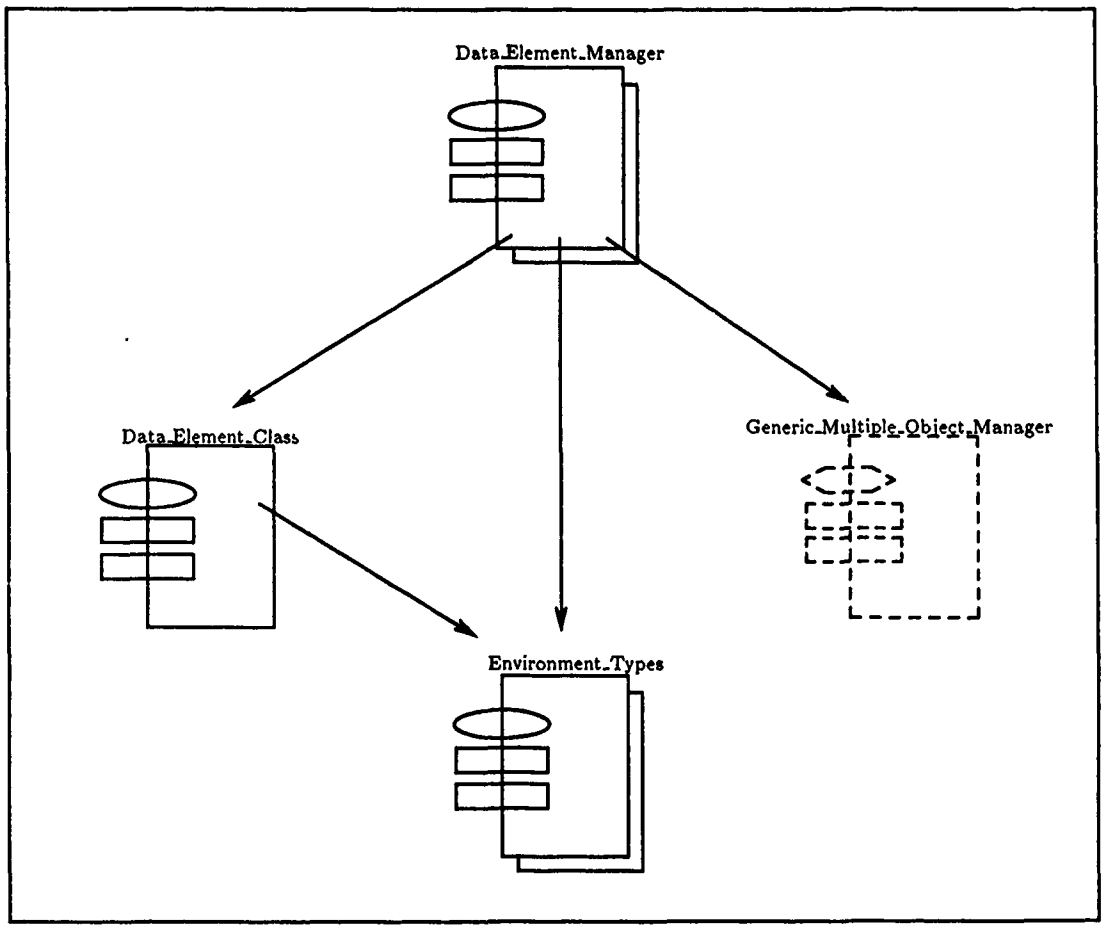


Figure 34. Module Diagram for Data_Element_Manager

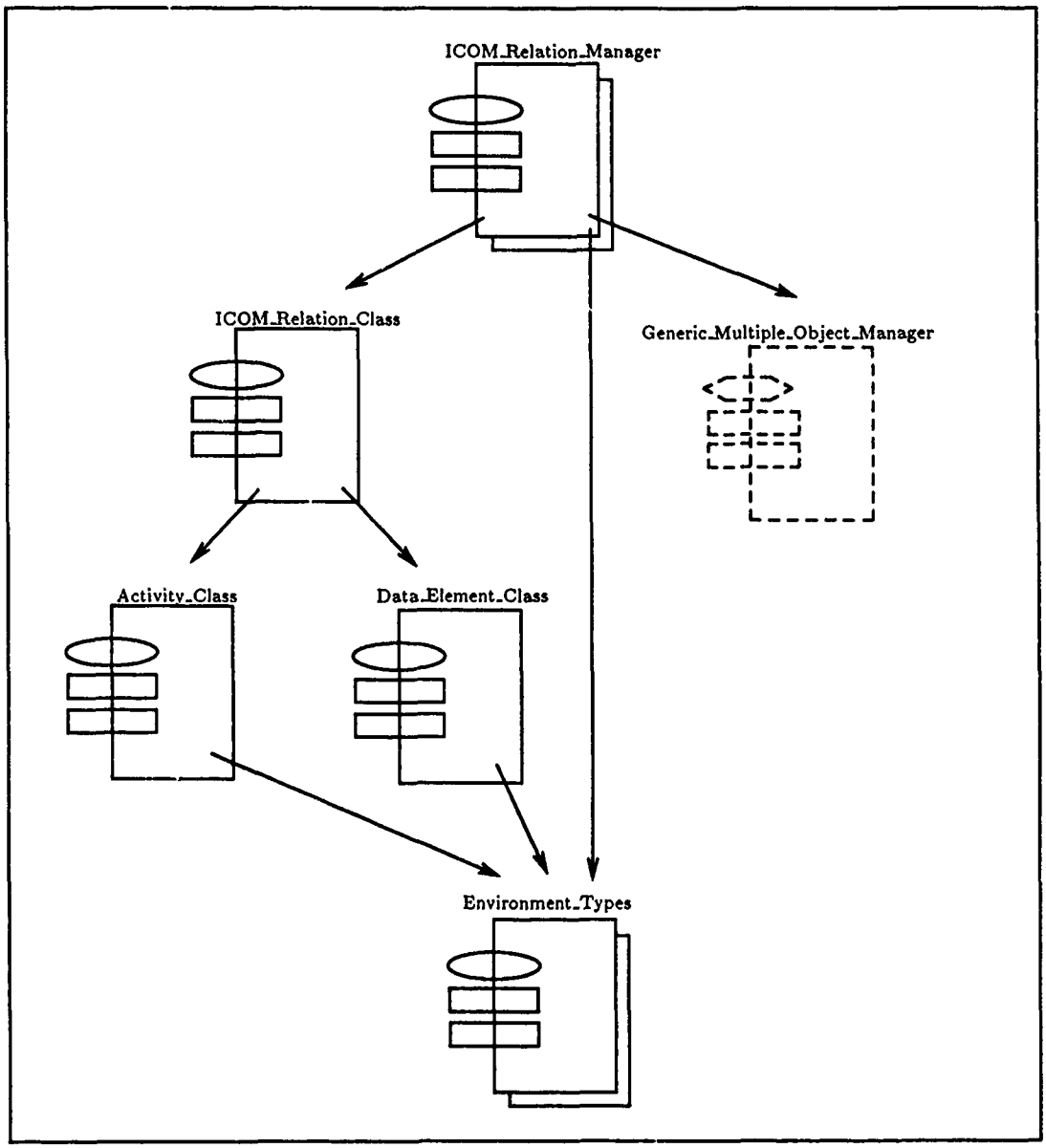


Figure 35. Module Diagram for ICOM_Relation_Manager

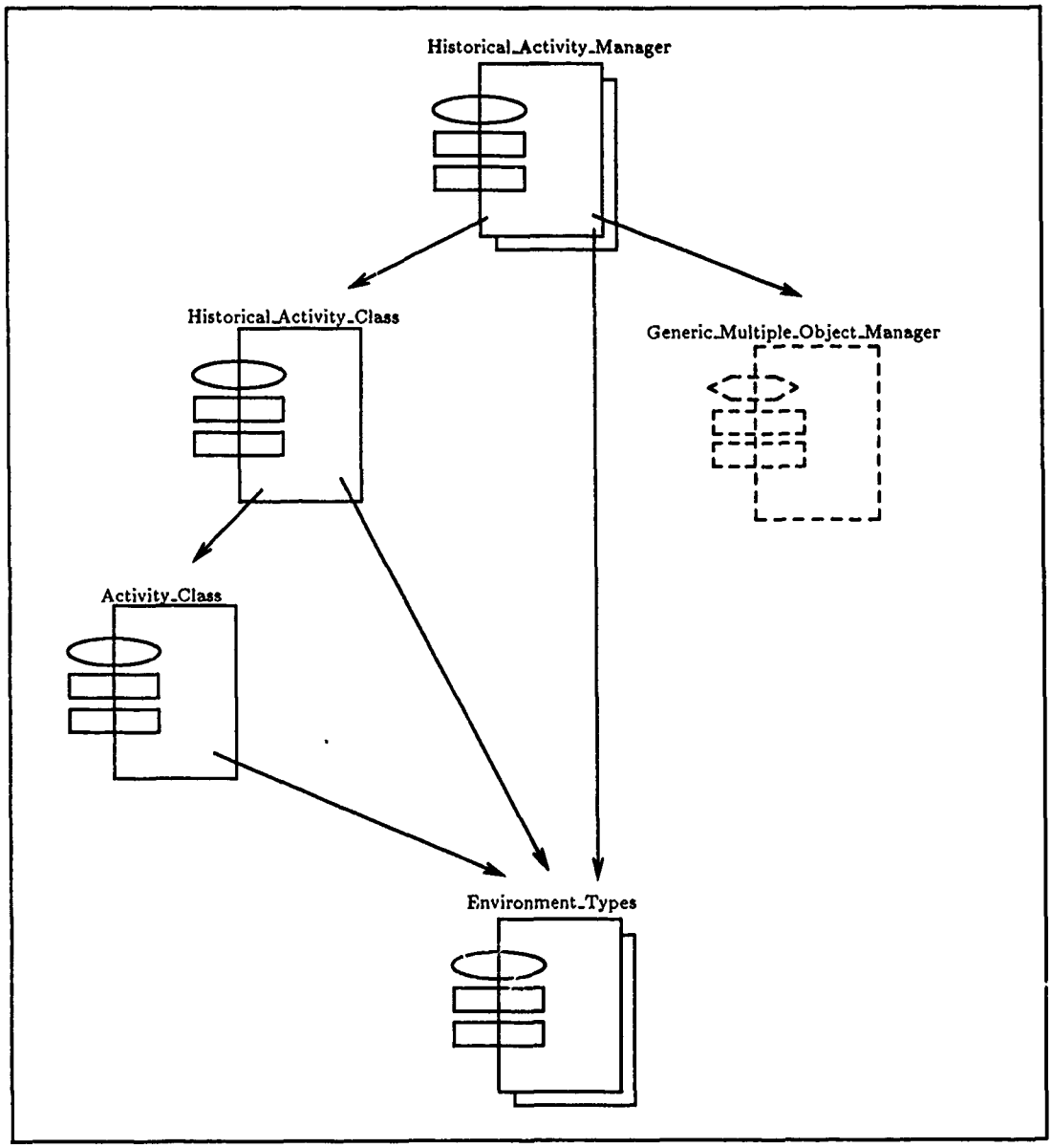


Figure 36. Module Diagram for Historical_Activity_Manager

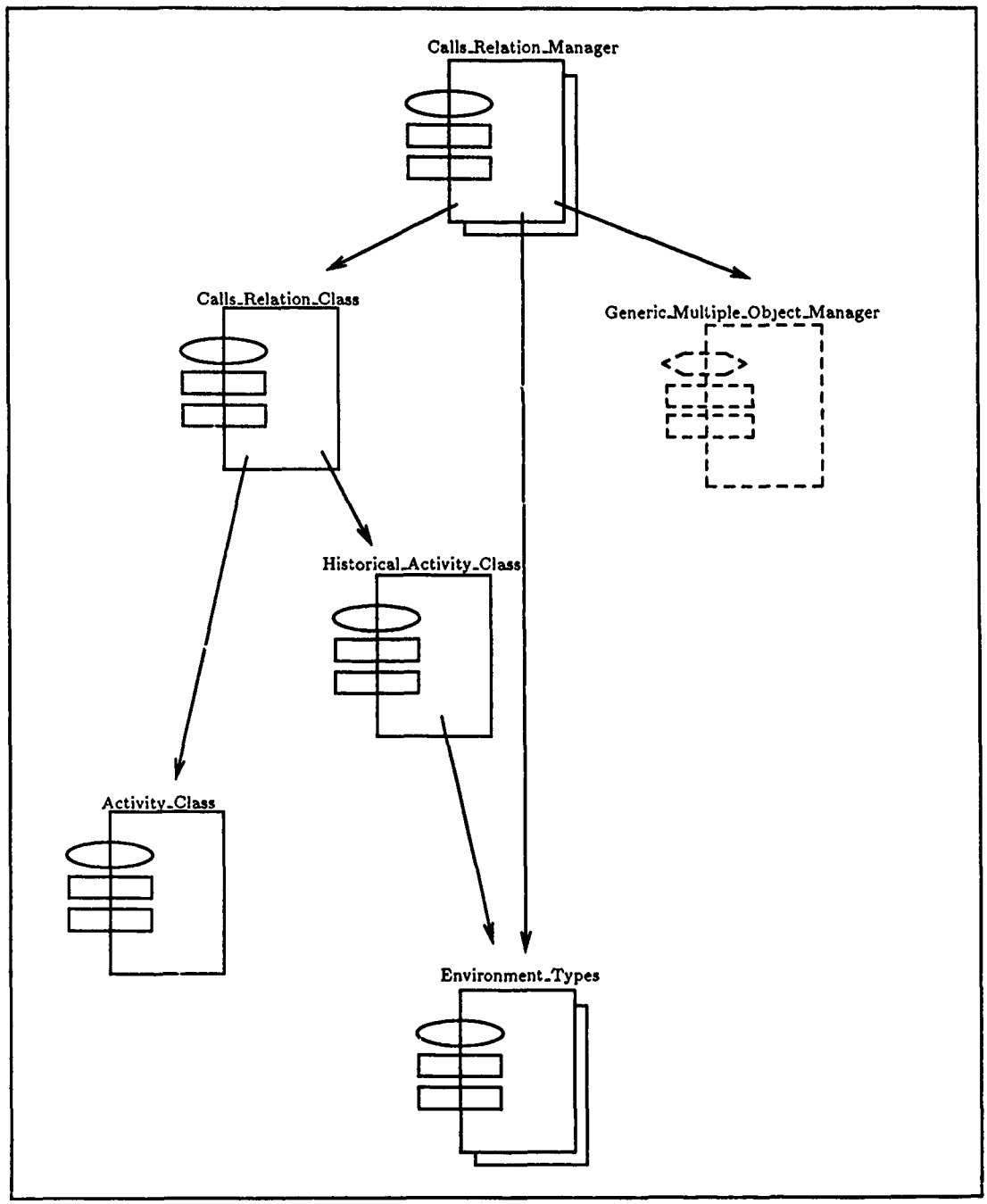


Figure 37. Module Diagram for Calls_Relation_Manager

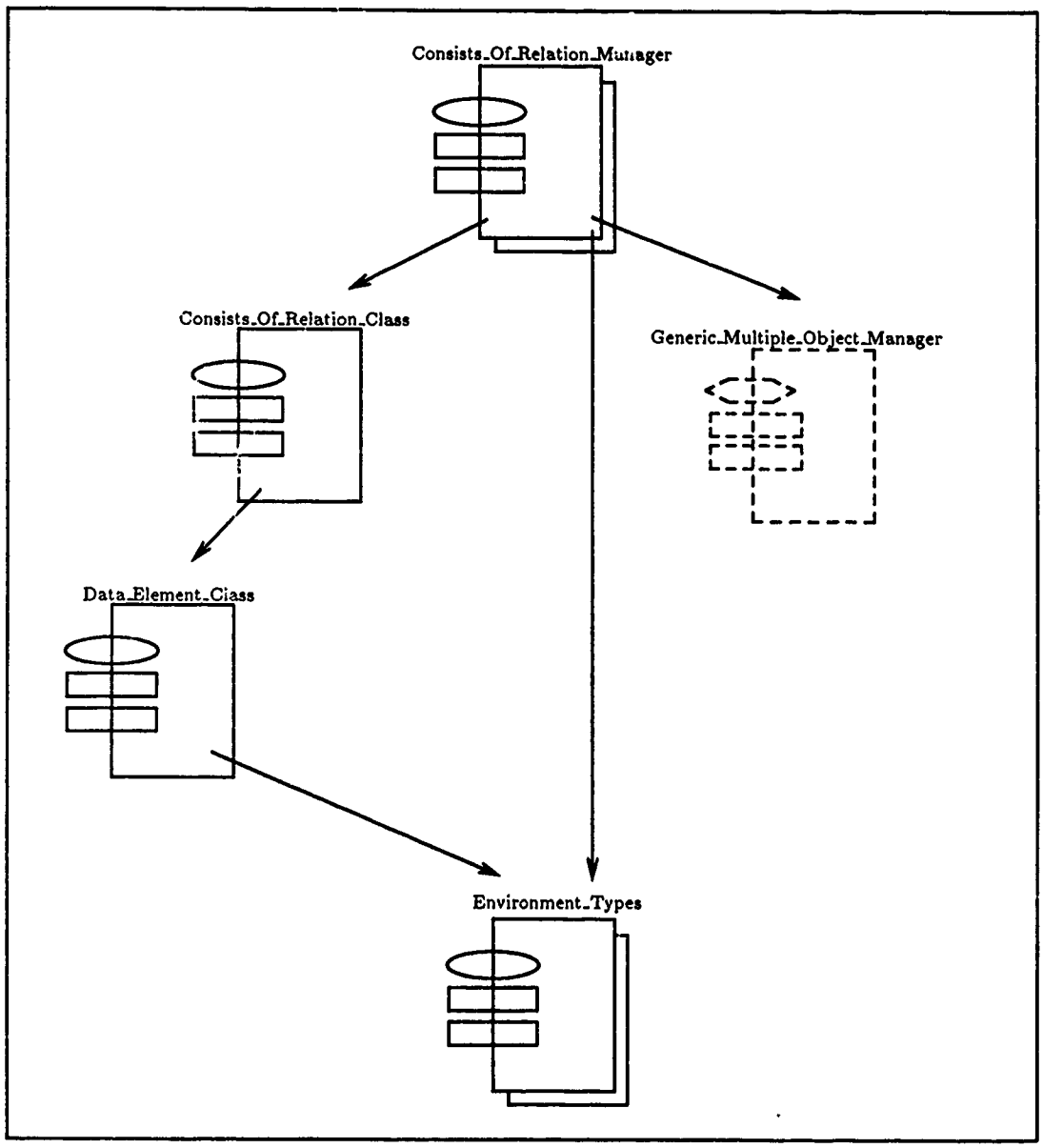


Figure 38. Module Diagram for Consists_Of_Relation_Manager

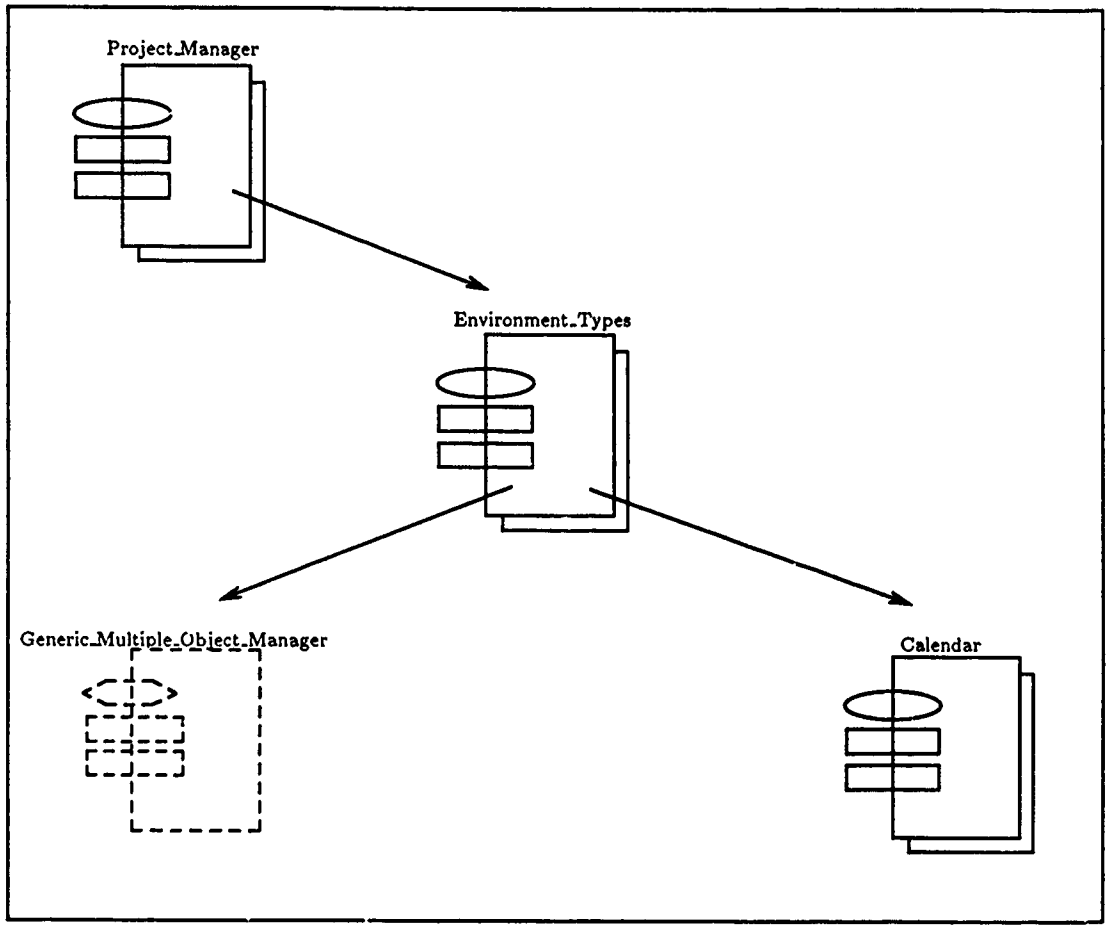


Figure 39. Module Diagram for Project_Manager and Environment_Types

Appendix B. *ESSENTIAL SUBSYSTEM CONFIGURATION GUIDE*

This appendix presents information concerning both the source code and configuration of the Essential Subsystem. The source code for the Essential Subsystem (excluding the CLIPS/Ada source code) can be found in Volume II of this thesis which is maintained by the Air Force Institute of Technology, Department of Electrical and Computer Engineering. See Appendix C for information concerning the CLIPS/Ada source code.

Of special note is the amount of disk space required for the Essential Subsystem. A total of 8 to 8.5 megabytes of free disk space is required for the Essential Subsystem to compile. This includes space for all the source code (including the CLIPS/Ada source code), the Ada library files created during compilation, and the Essential Subsystem executable. The Essential Subsystem executable is currently approximately 1.3 megabytes in size. However, much of this size can be attributed to the CLIPS/Ada expert system code.

To facilitate and document the compilation order for the Essential Subsystem, a shell program is developed to compile all the modules of the Essential Subsystem. The following shell program, 'shell_compile_satool2', not only documents the compilation order but also the contents of each of the source code files of the Essential Subsystem. Note the warning contained in the shell file that requires the CLIPS/Ada source code to have already been compiled.

```
# This is a unix shell file to compile the Essential Subsystem
# code.  The following command invokes the compilation:
# Execute this file by typing "csh shell_compile_satool2".
#
# Warning: In order for successful compilation to occur,
# the CLIPS/Ada source code must have already been compiled.
# A separate shell program performs that function.  The code
# must be compiled, because the package Clips_Working_Memory_
# Interface in the file es_clpwm.a requires the subunit
# Embedded_Clips which is part of the Clips/Ada source code.
#
# The names and contents of the files included in this shell are:
#
```

```
# es_genev.a    Package Generic_Multiple_Object_Manager
#               Package Environment_Types
# es_activ.a    Package Activity_Class
#               Package Activity_Manager
# es_datel.a    Package Data_Element_Class
#               Package Data_Element_Manager
# es_proj.a     Package Project_Manager
# es_hista.a    Package Historical_Activity_Class
#               Package Historical_Activity_Manager
# es_calls.a    Package Calls_Relation_Class
#               Package Calls_Relation_Manager
# es_conof.a    Package Consists_Of_Relation_Class
#               Package Consists_Of_Relation_Manager
# es_ICOM.a     Package ICOM_Relation_Class
#               Package ICOM_Relation_Manager
# es_factu.a    Package Essential_Fact_Utilities
# es_clpwm.a    Package Clips_Working_Memory_Interface
# es_esmio.a    Package Essential_IO
#
# ****These files contain modules used for testing and
# ****demonstration purposes only:
#
# es_mnuio.a    Package Menu_IO
# es_main.a     Subprogram es_main
#
# **************START THE EXECUTABLE COMMANDS**************
# This is the compilation order:
#
# Level 1 files:
echo Compiling Level 1 file
ada es_genev.a
#
# Level 2 files: These files can be compiled in any order.
echo Compiling Level 2 files
ada es_activ.a
ada es_datel.a
ada es_proj.a
#
# Level 3 files: Compile in any order.
echo Compiling Level 3 files
ada es_hista.a
ada es_ICOM.a
ada es_conof.a
ada es_mnuio.a
#
# Level 4 file:
echo Compiling Level 4 file
ada es_calls.a
#
# Level 5 file:
echo Compiling Level 5 file
```

```
ada es_factu.a
#
# Level 6 files: Compile in any order.
echo Compiling Level 6 files
ada es_esmio.a
ada es_clpwm.a
#
# Level 7 file: Compile last.
echo Compiling Main program
ada es_main.a
ada -M es_main.a
```

Appendix C. *CLIPS/ADA CONFIGURATION GUIDE*

CLIPS/Ada is an Ada version of CLIPS which has all the original functionality of CLIPS with a few exceptions. These exceptions and other information concerning CLIPS/Ada are contained in a "README.TXT" file that comes with the CLIPS/Ada source code. Although implemented in Ada, the design of CLIPS/Ada is strictly functional (33:2). CLIPS/Ada is available for free to any U.S. Government agency or government contractors by calling the CLIPS Help Desk at (713)280-2233. It is available outside of the U.S. Government and contractors through COSMIC, the NASA software distribution center.

To become familiar with CLIPS and CLIPS/Ada, four manuals should be acquired by calling the CLIPS Help Desk:

1. CLIPS User's Guide (35)
2. CLIPS Reference Manual (34)
3. CLIPS/Ada Architecture Manual (33)
4. CLIPS/Ada Advanced Programming Guide (32)

The source code for CLIPS/Ada was obtained via tape on reel #9446 which is now maintained by AFIT/SC. The tape contains CLIPS/Ada source code in ASCII form, a "README.TXT" file, some sample knowledge bases, and two .COM files (COMPILE.COM and LINK.COM) to compile and link the source code. It also contains an executable version of CLIPS/Ada. However, a VAX Ada compiler version 1.5 running under VAX-VMS 5.1.1 is used to create the executable. Therefore, it is useless on Unix based machines.

Since the primary platform for this research is a SUN-3 running a version of UNIX and a Verdix Ada compiler, several changes to the original source code are necessary. First, all the CLIPS/Ada source code files had .ADA or .ADS extensions that are unacceptable to Verdix. All

.ADS files were changed to "_spec.a" files, and all .ADA files were changed to .a files. Once all the names were changed, the code was transferred to Olympus and compiled. Several syntax errors relating to incompatibility problems between VAX Ada and Verdix Ada were corrected. However, many warning messages are still received when compiling CLIPS/Ada. These messages are due to the source code authors explicitly declaring loop counters. VAX Ada obviously allows such declarations; Verdix Ada allows them also but does not particularly care for them. Therefore, Verdix Ada issues a warning message.

In order to facilitate CLIPS/Ada file compilation, five shell files are created. The execution of the shell file "shell_compile_all_CLIPS" results in the compilation of all the CLIPS/Ada files necessary for the Essential Subsystem compilation. To be specific, these files must be compiled prior to the compilation of the CLIPS_Working_Memory_Interface package. By executing the aforementioned shell file, all the CLIPS/Ada source code is compiled with the exception of one file - Clips_Ada.a. Clips_Ada.a is the main routine for creating the CLIPS/Ada expert system shell alone. If the CLIPS/Ada expert system shell alone is desired, simply execute the aforementioned shell program first, followed by "ada -M Clips_Ada.a".

In order to document the necessary compilation order for the CLIPS/Ada source code, five shell programs are illustrated. The command 'csh shell_compile_all_CLIPS' is the only command that needs to be executed, because it invokes the four other shell programs.

This is the shell program 'shell_compile_all_CLIPS' which controls the compilation of all the CLIPS/Ada source code. Note that the execution of this shell program will result in the addition of approximately 3.2 megabytes of disk space to the directory from which it is executed. This is due to the creation of the Ada library files.

```
# This shell file makes 4 calls to other shell programs.
# Start this shell by typing> csh shell_compile_all_CLIPS
# This shell compiles all the files required to use CLIPS/Ada
# as an embedded expert system.  When this is done, you can
```

```
# can do "with embedded_clips;" in the client program. 13 Sep 90, TLK.
#
# NOTE: You will get 30 or so compiler WARNING messages.
# You can ignore them.  The code was written for a VMS Ada
# compiler which doesn't mind if you declare a variable like "counter"
# and then use it later in a stmt like:   for counter in 1..10 loop.
# However, Verdix Ada doesn't like that; therefore, you may get
# warning messages depending on the Ada compiler used.
csh shell_compile_specs
csh shell_compile_subunits
csh shell_compile_bodies
csh shell_compile_separates
```

This is the shell program 'shell_compile_specs':

```
# Shell to compile the CLIPS/Ada package specifications.
ada globals_spec.a
ada analysis_spec.a
ada buffer_spec.a
ada clips_spec.a
ada clipsio_spec.a
ada deffacts_spec.a
ada engine_spec.a
ada envmnt_spec.a
ada evaluate_spec.a
ada expressn_spec.a
ada factmngr_spec.a
ada generate_spec.a
ada lhsparse_spec.a
ada math_spec.a
ada mathex_spec.a
ada memory_spec.a
ada netmngr_spec.a
ada ppclips_spec.a
ada reorder_spec.a
ada rulemngr_spec.a
ada rulelist_spec.a
ada scanner_spec.a
ada strings_spec.a
ada symbol_spec.a
ada sysio_spec.a
ada sysprime_spec.a
ada syssecnd_spec.a
ada userfun_spec.a
ada utility_spec.a
ada variable_spec.a
ada dbugfunc_spec.a
ada envnfunc_spec.a
```

```
ada memrfunc_spec.a
ada rulefunc_spec.a
ada factfunc_spec.a
ada netwfunc_spec.a
ada strgfunc_spec.a
ada strgutil_spec.a
ada mathlib_spec.a
ada dtmngr_spec.a
ada dtfunc_spec.a
ada charutil_spec.a
ada multfunc_spec.a
```

This is the shell program 'shell_compile_subunits'. Note that this shell program compiles the file 'emclips_spec.a'. This file contains the specification for the package 'Embedded_Clips'. It is this package that a client program must "with in" in order for the client to be able to directly access the contents of the working memory of the CLIPS/Ada expert system.

```
# Shell to compile the CLIPS/Ada subunits.  The first two are
# generics.  The second two are specs.  emclips_spec.a contains
# the spec that is used to embed CLIPS/Ada in another program.
# They are placed in this separate file only because the COMPILE.COM
# file called these sub-units which distinguished them from the
# remaining specs and bodies.  Why?  I'm not sure.
ada enumcvrt.a
ada evalgens.a
ada syspred_spec.a
ada emclips_spec.a
```

This is the shell program 'shell_compile_bodies':

```
# Shell to compile the CLIPS/Ada package bodies.
ada analysis_body.a
ada buffer_body.a
ada variable_body.a
ada clips_body.a
ada clipsio_body.a
ada deffacts_body.a
ada emclips_body.a
ada engine_body.a
ada envmnt_body.a
ada evaluate_body.a
ada expressn_body.a
```

```
ada factmngr_body.a
ada generate_body.a
ada lhsparse_body.a
ada math_body.a
ada mathex_body.a
ada memory_body.a
ada netmngr_body.a
ada ppclips_body.a
ada reorder_body.a
ada rulemngr_body.a
ada rulelist_body.a
ada scanner_body.a
ada strings_body.a
ada symbol_body.a
ada sysio_body.a
ada syspred_body.a
ada sysprime_body.a
ada syssecnd_body.a
ada userfun_body.a
ada utility_body.a
ada dbugfunc_body.a
ada envnfunc_body.a
ada memrfunc_body.a
ada rulefunc_body.a
ada factfunc_body.a
ada netwfunc_body.a
ada strgfunc_body.a
ada strgutil_body.a
ada mathlib_body.a
ada dtmngr_body.a
ada dtfunc_body.a
ada charutil_body.a
ada multfunc_body.a
```

This is the shell program 'shell_compile_separates':

```
# Shell file to compile the CLIPS/Ada 'separate' subprograms.
ada drivefct.a
ada joincomp.a
ada matchret.a
ada buildnet.a
ada pttncomp.a
ada rmrulent.a
ada compfact.a
ada showjn.a
ada showflks.a
```

Appendix D. *CLIPS RULE BASE*

This appendix presents the CLIPS rule base that was used to perform the syntax checking for $IDEF_0$ models. Note that these rules are but a subset of the rules required to fully check the syntax of an $IDEF_0$ model.

This file is called 'satool2.clp' and is separate from the SAtool II object code. It must be available in the current path in order to perform the syntax check function contained in the Essential Subsystem menu.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;               Essential Subsystem Rule Base                ;;
;; File Name: satool2.clp                                     ;;
;; Date Last Updated: 12 Nov 90                               ;;
;; Author: Terry Kitchen, GCS-90D                             ;;
;; Points of Contact: Dr. Thomas Hartrum, Dr. Gary Lamont     ;;
;; Description:                                               ;;
;;     This file contains the rule base used by the           ;;
;; the CLIPS/Ada expert system portion of the Essential       ;;
;; Subsystem.  This subsystem is to eventually be integrated  ;;
;; with another system to form SAtool II, which is an Ada     ;;
;; based IDEFO development tool.                              ;;
;; Purpose:                                                   ;;
;;      The purpose of this rule base is to check the         ;;
;; syntactic features of an IDEFO model whose representation  ;;
;; has been converted to CLIPS readable facts.                ;;
;; Methodology:                                               ;;
;;      Whenever the "check syntax" option is chosen within   ;;
;; the Essential Subsystem main menu, this rule base is loaded;;
;; into the working memory of the CLIPS/Ada expert system.    ;;
;; The same option also begins the "recognize-act" cycle of   ;;
;; the CLIPS inference engine which uses the rules below to   ;;
;; "match" the LHS of rules with facts, resolve conflicts     ;;
;; among eligible rules, and then fire the RHS of rules, until;;
;; no rules are eligible to fire.  This file must be within   ;;
;; Scope:                                                     ;;
;;      At the present time, this rule base only checks the   ;;
;; syntactical features associated with the "essential" data  ;;
;; of an IDEFO model.  This rule base only checks a very      ;;
;; limited number of features of an IDEFO model.  This is due ;;
;; primarily to the time constraints imposed on the research  ;;
;; which prevented completion of all the translation routines ;;
;; within the Essential_Fact_Utilities package of the         ;;
;; Essential Subsystem.                                       ;;
```

```
;; Features Checked:                                                ;;
;;      The following subset of IDEF0 features are checked:        ;;
;; 1) Each activity is checked to ensure it has at least one       ;;
;;    control.                                                     ;;
;; 2) Each activity is checked to ensure it has at least one       ;;
;;    output.                                                      ;;
;; 3) Each activity is checked to ensure it has an activity        ;;
;;    number.                                                      ;;
;; 4) Each activity is checked to ensure it has a description.    ;;
;; 5) The current IDEF0 model under development must have a        ;;
;;    project name.  It cannot be null.                            ;;
;; Output:                                                         ;;
;;      IDEF0 syntax violations cause the user to receive          ;;
;; "error" messages.  The lack of a description for an             ;;
;; activity warrants just a "warning" message, since it is         ;;
;; not part of the IDEF0 diagram itself.  If no "errors" are       ;;
;; detected, a congratulatory message is output.                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "print-intro" precedes any syntax error, or
;; warning messages.  This is guaranteed by the salience.
(defrule print-intro
 (declare (salience  10) )
 (initial-fact)
=>
 (printout t crlf "****Essential Subsystem Syntax Messages****" crlf))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RULE FOR CHECKING PROJECT NAME
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "null-project-name" verifies that the current IDEF0
;; model under development has been assigned a name.  Note that
;; this rule can be removed if the final implementation of SAtool II
;; forces the user to always enter a project name.
(defrule  null-project-name
 (project-name  null)
=>
 (printout t "ERROR: The current project does not have a name." crlf)
 (assert (syntax-error-occurred)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RULE FOR CHECKING ACTIVITY NUMBER
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "null-activity-number" will fire if user has not
;; assigned an activity number to any of the activities.
(defrule  null-activity-number
 (act-numb  ?activity   null)
=>
 (printout t "ERROR: Activity " ?activity " must be numbered." crlf)
```

```
 (assert (syntax-error-occurred)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RULE FOR CHECKING ACTIVITY DESCRIPTION
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "null-activity-description" will fire if user has not
;; assigned a description to an activity.  Unlike other violations,
;; this rates only a warning message if missing.
(defrule  null-activity-description
 (act-desc  ?activity   null)
=>
 (printout t "WARNING: Activity " ?activity " needs a description." crlf))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RULE FOR CHECKING OUTPUT ARROWS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "zero-outputs" checks each activity to make sure it
;; does not have 0 outputs.  If so, it is an error.
(defrule  zero-outputs
 (icom-activity-outputs  ?act  0)
=>
 (printout t "ERROR: Activity " ?act " needs at least 1 output." crlf)
 (assert (syntax-error-occurred)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RULE FOR CHECKING CONTROL ARROWS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "zero-controls" checks each activity to make sure it
;; does not have 0 controls.  If it has no controls, it is an error.
(defrule  zero-controls
 (icom-activity-controls  ?act  0)
=>
 (printout t "ERROR: Activity " ?act " needs at least 1 control." crlf)
 (assert (syntax-error-occurred)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TERMINATION RULES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The rule "exit-if-error" is the next to last rule that
;; can possibly be fired.  If any errors have occurred, this rule
;; insures the expert system stops at this point.
(defrule exit-if-error
 (declare (salience  -1) )
 (syntax-error-occurred)
=>
 (halt))
```

```
;; The rule "no-errors-encountered" is the last rule that
;; can possibly be fired, because of its low salience.  This
;; rule will fire only if no errors have been encountered.
(defrule no-errors-encountered
 (declare (salience  -2) )
 ?flag  <- (initial-fact)
=>
 (retract  ?flag)
 (printout t "CONGRATULATIONS: No syntax errors encountered." crlf))
;;;;;;;;;;;;;;;;;;;;;;;;;;;END OF RULE BASE;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Appendix E. *PACKAGE AND SUBROUTINE HEADERS*

This appendix presents the revised subroutine header format and the new package header format that are used to document the Essential Subsystem source code. Presented first is the revised Module/Subr··tine/Procedure header format based on (16:34).

Since the term "module" is now used to refer to packages (7), it is recommended that the word "module" be removed from the header name and the header format as well. For example, "MODULE NUMBER" could become "SUBROUTINE NUMBER" or even "SUBPROGRAM NUMBER". The package header can then be called a Module/Package header.

```
------------------------------------------------------------------------
-- DATE: mm/dd/yy                                                      --
-- VERSION:                                                            --
-- NAME:                                                               --
-- MODULE NUMBER:                                                      --
-- DESCRIPTION:                                                        --
-- ALGORITHM: is some "well known" algorithm used? quicksort e.g.      --
-- If so, state it; otherwise briefly discuss the methodology: loops,  --
-- case statements, etc.                                               --
-- PASSED VARIABLES:                                                   --
-- RETURNS:                                                            --
-- GLOBAL VARIABLES USED:                                              --
-- GLOBAL VARIABLES CHANGED:                                           --
-- FILES READ:                                                         --
-- FILES WRITTEN:                                                      --
-- HARDWARE INPUT:                                                     --
-- HARDWARE OUTPUT:                                                    --
-- MODULES CALLED:                                                     --
-- CALLING MODULES:                                                    --
-- ORDER-OF:  State the order of in Big-O notation.  Embed analysis    --
--            within the source code.                                  --
-- AUTHOR(S):                                                          --
-- HISTORY:                                                            --
------------------------------------------------------------------------
```

This is the format developed and used for package headers.

```
-------------------------------------------------------------------------
-- DATE: mm/dd/yy                                                       --
-- VERSION:                                                             --
-- PACKAGE NAME:                                                        --
-- LOCATED IN FILE:                                                     --
-- PURPOSE: Is the pkg an ADT, Abstract State Machine, a group of       --
-- operations?  What does it provide?                                   --
-- PACKAGE VISIBILITIES REQUIRED: What is with'd in?                    --
-- PACKAGE COMPOSITION: Is there a spec and a body or just a spec?      --
-- GENERICS INSTANTIATED:  List packages instantiated in spec or body.--
-- ADT DESCRIPTION:  ONLY applicable if this is an Abstract Data Type.--
-- Follow examples in "Data Structures With Pascal" by Horowitz and     --
-- Sahni.  The domain and operations are mandatory but the axioms       --
-- are optional.                                                        --
-- This is a sample entry:                                              --
-- ADT FOR NARY_TREE:                                                   --
-- Structure NARY_TREE(tree, natural, boolean, item)                    --
--   Declare                                                            --
--   *Create()  =>                                tree                  --
--    Clear(tree) =>                              tree                  --
--    Construct(tree, item, natural, natural) => tree'                  --
--    Set_Item(tree, item) =>                     tree'                 --
--    Swap_Child(natural, tree, tree') =>         tree''                --
--    Is_Equal(tree, tree')  =>                   boolean               --
--    Is_Null(tree)  =>                           boolean               --
--    Item_Of(tree) =>                            item                  --
--    Number_Of_Children_In(tree) =>              natural               --
--    Child_Of(tree, natural) =>                  tree'                 --
-- end                                                                  --
-- END NARY_TREE                                                        --
-- *Note: Create is implemented in Ada by instantiation of              --
-- the generic package and subsequent variable declaration.             --
--                                                                      --
-- ORDER-OF:  Simply list each procedure and function included in the   --
-- package in a single column with its order-of next to it.  Separate   --
-- the list into 2 groups - those visible and those not visible.        --
-- Only list procedures and functions whose modules are in the spec     --
-- and body.  Any "separate" modules should probably be flagged as      --
-- such somehow.  If order-ofs come from a text, site the source.       --
-- This is a sample entry:                                              --
-- Visible:   Copy                    O(n)                              --
--            Clear                   O(1)                              --
--            Construct               O(1)                              --
--            Set_Item                O(1)                              --
--            Swap_Child              O(1)                              --
--            Is_Equal                O(n)                              --
--            Is_Null                 O(1)                              --
```

```
--            Item_Of                  O(1)                              --
--            Number_Of_Children_In    O(1)                              --
--            Child_Of                 O(1)                              --
-- Hidden: (none)                                                        --
-- Where n is the number of elements in the tree.                        --
-- AUTHOR(S):                                                            --
-- HISTORY:                                                              --
--------------------------------------------------------------------------
```

Appendix F. *SAMPLE IDEF$_0$ MODEL OUTPUT FILE*

This appendix presents a sample output file for an IDEF$_0$ project. The format of this file is a CLIPS readable fact base. The data in the file is based on Figures 40 and 41 which represent a sample IDEF$_0$ model (project). Note that the data contained in the file is only a small subset that would actually be required to represent this diagram in its entirety. Specifically, all the facts associated with both the Project Manager and ICOM Relation Manager are stored. Also, the activity number, activity description, activity name, and parent-child relationships from the Activity Manager are stored. No state information from the remaining four managers is retrieved.

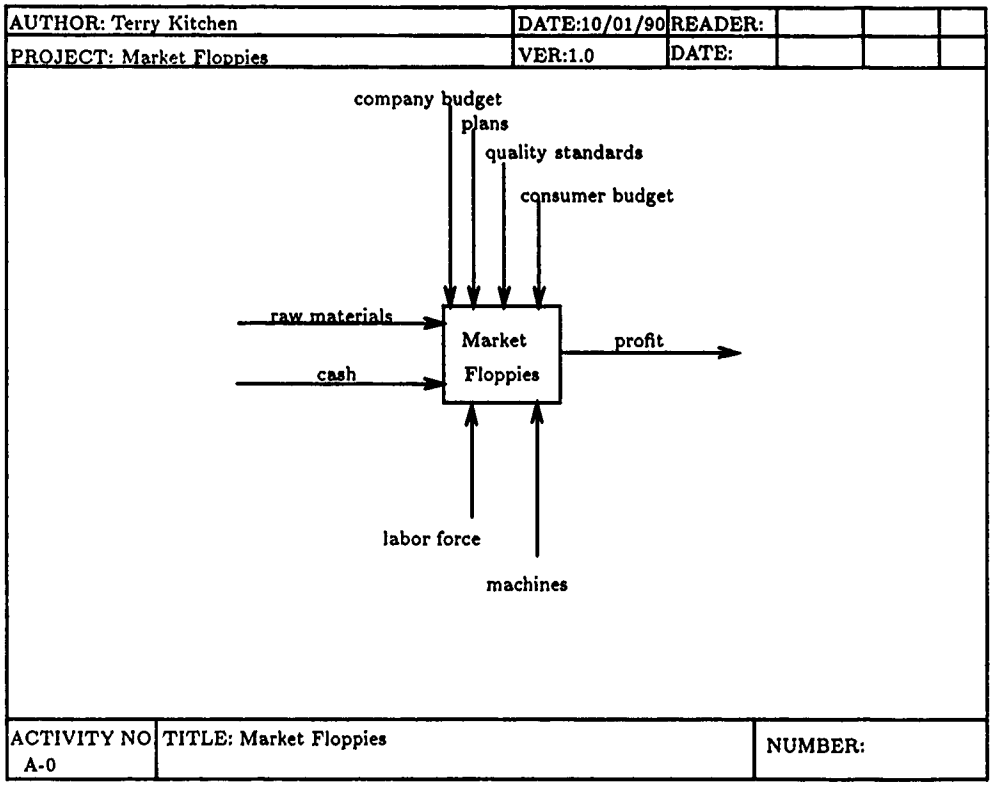


Figure 40. A-0 Diagram for 'Market Floppies'

Assuming that Figures 40 and 41 have been entered into the Essential Subsystem via the Essential Subsystem Test and Demonstration Program, the following output file contains the state information of the 'Market Floppies' project:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SAtool II - IDEF0 Essential Fact File - CLIPS Readable Format
;; Date and Time of File Creation : 11/17/90 21:20:15
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;**START ALL FACTS**
(deffacts icom-facts
(icom-tuple Market_Floppies          company_budget          c     1)
(icom-tuple Market_Floppies          profit                  o     2)
(icom-tuple Buy_Floppies             company_budget          c     3)
(icom-tuple Sell_in_Store_A          consumer_budget         c     4)
(icom-tuple Manufacture_Floppies     company_budget          c     5)
(icom-tuple Market_Floppies          plans                   c     6)
(icom-tuple Market_Floppies          quality_standards       c     7)
(icom-tuple Market_Floppies          consumer_budget         c     8)
(icom-tuple Market_Floppies          raw_materials           i     9)
(icom-tuple Market_Floppies          cash                    i    10)
(icom-tuple Market_Floppies          labor_force             m    11)
(icom-tuple Market_Floppies          machines                m    12)
(icom-tuple Buy_Floppies             cash                    i    13)
(icom-tuple Buy_Floppies             floppies                o    14)
(icom-tuple Sell_in_Store_A          floppies                i    14)
(icom-tuple Sell_in_Store_A          profit                  o    15)
(icom-tuple Manufacture_Floppies     raw_materials           i    16)
(icom-tuple Manufacture_Floppies     quality_standards       c    17)
(icom-tuple Manufacture_Floppies     plans                   c    18)
(icom-tuple Manufacture_Floppies     labor_force             m    19)
(icom-tuple Manufacture_Floppies     machines                m    20)
(icom-tuple Manufacture_Floppies     floppies                o    21)
(icom-tuple Sell_in_Store_B          floppies                i    21)
(icom-tuple Sell_in_Store_B          consumer_budget         c    22)
(icom-tuple Sell_in_Store_B          profit                  o    23)
)
(deffacts project-facts
(project-name Market_Floppies)
)
(deffacts activity-facts
(act-name Market_Floppies)
(act-numb Market_Floppies          A0)
(act-desc Market_Floppies          The company's business is to market)
(act-desc Market_Floppies          five and one quarter inch floppy)
(act-desc Market_Floppies          disks.)
(act-has-child Market_Floppies          Buy_Floppies)
(act-has-child Market_Floppies          Sell_in_Store_A)
(act-has-child Market_Floppies          Manufacture_Floppies)
(act-has-child Market_Floppies          Sell_in_Store_B)
(act-name Buy_Floppies)
(act-numb Buy_Floppies             A1)
(act-desc Buy_Floppies             Based on the company budget, use cash)
(act-desc Buy_Floppies             on hand to buy floppies from an)
(act-desc Buy_Floppies             outside source.)
(act-has-child Buy_Floppies          null)
```

```
(act-name Sell_in_Store_A)
(act-numb Sell_in_Store_A          A2)
(act-desc Sell_in_Store_A          Sell the floppies bought from an)
(act-desc Sell_in_Store_A          outside source in Store A only.)
(act-has-child Sell_in_Store_A          null)
(act-name Manufacture_Floppies)
(act-numb Manufacture_Floppies     A3)
(act-desc Manufacture_Floppies     Using our own manufacturing resources)
(act-desc Manufacture_Floppies     and standards, make floppy disks.)
(act-has-child Manufacture_Floppies     null)
(act-name Sell_in_Store_B)
(act-numb Sell_in_Store_B          A4)
(act-desc Sell_in_Store_B          Sell the floppies that we make in)
(act-desc Sell_in_Store_B          Store B only.)
(act-has-child Sell_in_Store_B          null)
)
;;**END ALL FACTS**
```

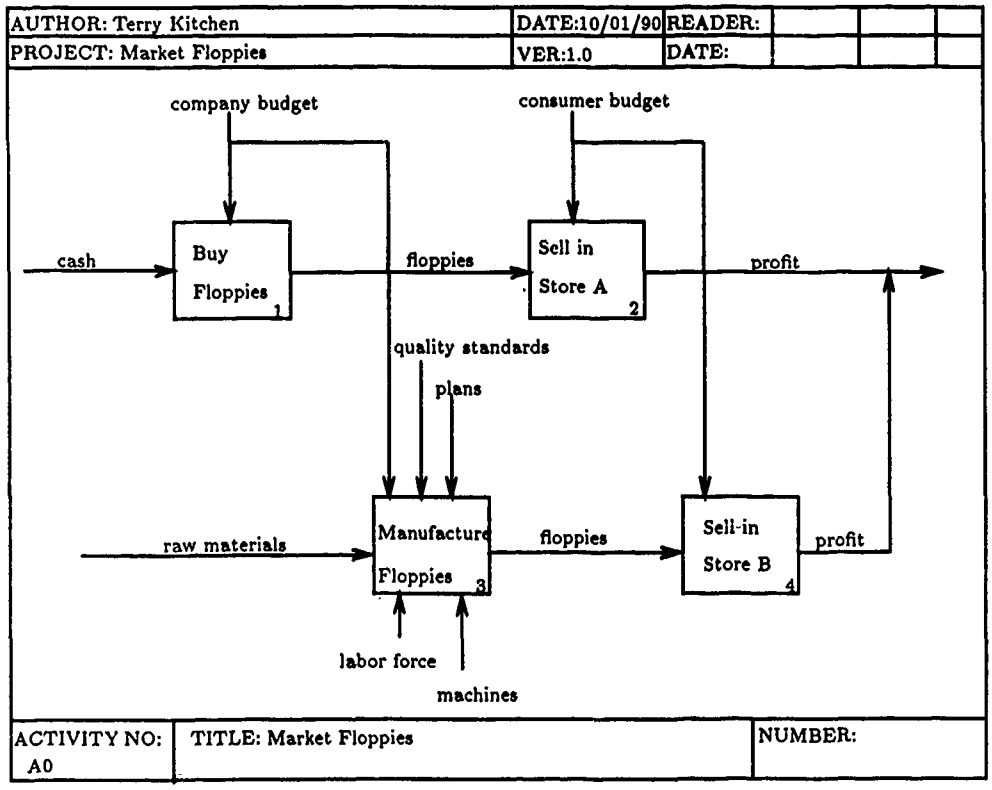


Figure 41. A0 Diagram for 'Market Floppies'

Appendix G. *ESSENTIAL SUBSYSTEM TEST AND DEMONSTRATION PROGRAM SCRIPT*

This appendix performs several purposes:

1. It demonstrates some of the functionality and menus of the Essential Subsystem Test and Demonstration Program.
2. It illustrates how facts are stored in the CLIPS/Ada Working Memory.
3. It illustrates the results of performing a syntax check on a sample $IDEF_0$ model.

In this script, the Essential Subsystem loads a project very similar to the sample $IDEF_0$ project presented in Appendix F. This project has the same project name but is stored in a different file – 'demo_syntax_errors.esm'. The only difference in the two projects is that the activity 'Sell in Store B' does not have an activity number or description assigned to it. It also has no control or output arrows. These features can be seen by examining the facts associated with 'Sell in Store B' in the script file.

There two notable differences between the way the facts in the '.esm' file are stored versus the facts used for syntax checking.

- The activity description is not stored. Only the words "null" or "not-null" are stored to represent their presence or absence in the model.
- Some additional facts are present that convey the number of inputs, outputs, controls, and mechanisms for each activity.

The script file that follows illustrates the initialization of the Essential Subsystem, followed by the loading of the file 'demo_syntax_errors.esm'. Next, the facts associated with the project are displayed. Finally, a syntax check is performed on the model, and the results output to the screen.

Of special note is the way that CLIPS stores all numbers as real numbers. This could be a potential problem when performing matches with rules. However, there was insufficient time to research this potential problem.

Of special note is the fact that the Essential Subsystem Test and Demonstration Program automatically replaces all blanks in any multi-character input with underscores. This is necessary because CLIPS recognizes blanks as delimiters between fields. Thus, the activity "Buy Floppies" is represented internally as "Buy_Floppies" to prevent CLIPS from separating the name into two fields.

```
Script started on Sat Nov 17 20:48:53 1990
csh> a.out
            CLIPS/Ada Version 4.30  10/12/89

 ******************************************************
 *              THE ESSENTIAL SUBSYSTEM               *
 *           TEST AND DEMONSTRATION MAIN MENU         *
 *            -- SAtool II Level Operations --        *
 ******************************************************
 Enter     To select the desired operation
   1.      Restore (load) a project from disk
           (Warning: all current data cleared)
   2.      Save the current project to disk
   3.      Display the current project name
   4.      Change the current project name
   5.      Create and display a data dictionary entry
   6.      Add a box/activity to the project
   7.      Connect 2 boxes with a data element/arrow
   8.      Check Syntax of current project
   9.      -- Submenus for Low Level Operations --
   0.      EXIT
SELECT A NUMBER: 1
Enter the file name of the project to be restored.
Do not include the file name extension.
Enter Name: demo_syntax_errors

Looking for essential data under file name: demo_syntax_errors.esm
Preparing to read facts from disk into a buffer.
A set of facts has been extracted from the file.
Calling procedure to load icom facts.
Procedure to restore ICOM facts done.
A set of facts has been extracted from the file.
Calling procedure to load project name fact.
```

```
Procedure to restore project name is done.
A set of facts has been extracted from the file.
Calling procedure to load activity facts.
Procedure to restore activity facts is done.
Project successfully restored.

PRESS ANY KEY - THEN RETURN TO CONTINUE:

 ******************************************************
 *              THE ESSENTIAL SUBSYSTEM               *
 *          TEST AND DEMONSTRATION MAIN MENU          *
 *           -- SAtool II Level Operations --         *
 ******************************************************
 Enter     To select the desired operation
   1.      Restore (load) a project from disk
           (Warning: all current data cleared)
   2.      Save the current project to disk
   3.      Display the current project name
   4.      Change the current project name
   5.      Create and display a data dictionary entry
   6.      Add a box/activity to the project
   7.      Connect 2 boxes with a data element/arrow
   8.      Check Syntax of current project
   9.      -- Submenus for Low Level Operations --
   0.      EXIT
SELECT A NUMBER: 9

******************************************************
*  ESSENTIAL SUBSYSTEM EDITING AND DEBUGGING MENU    *
* Warning: These operations allow you to directly    *
* exercise the object operations.  Use extreme care.*
* --Essential Model and Utility Level Operations--   *
******************************************************
 Enter     To select the desired submenu of operations
   1.      Activity Operations Menu
   2.      Data Element Operations Menu
   3.      Historical Activity Operations Menu
   4.      Calls Relation Operations Menu
   5.      ICOM Relation Operations Menu
   6.      Consists_Of Relation Operations Menu
   7       CLIPS Operations Menu
   8.      ICOM Fact Operations Menu
   9.      Activity Fact Operations Menu
   0.      EXIT
 SELECT A NUMBER: 7

  Enter To select this operation
    1.  Assert all facts into the CLIPS Working Memory.
    2.  Display all the facts in CLIPS Working Memory.
    3.  Clear the CLIPS Working Memory.
    0.  EXIT
```

```
SELECT A NUMBER: 1
ICOM facts for CLIPS retrieved, if any.
CLIPS WM - a set of facts were asserted.
Project name fact retrieved.
CLIPS WM - a set of facts were asserted.
Activity facts for CLIPS retrieved, if any.
CLIPS WM - a set of facts were asserted.
All facts for CLIPS retrieved.

PRESS ANY KEY - THEN RETURN TO CONTINUE:

 Enter To select this operation
   1.  Assert all facts into the CLIPS Working Memory.
   2.  Display all the facts in CLIPS Working Memory.
   3.  Clear the CLIPS Working Memory.
   0.  EXIT
SELECT A NUMBER: 2

*********Start of Working Memory********.
f-1     (icom-tuple Market_Floppies company_budget c 1)
f-2     (icom-tuple Market_Floppies profit o 2)
f-3     (icom-tuple Buy_Floppies company_budget c 3)
f-4     (icom-tuple Sell_in_Store_A consumer_budget c 4)
f-5     (icom-tuple Manufacture_Floppies company_budget c 5)
f-6     (icom-tuple Market_Floppies plans c 6)
f-7     (icom-tuple Market_Floppies quality_standards c 7)
f-8     (icom-tuple Market_Floppies consumer_budget c 8)
f-9     (icom-tuple Market_Floppies raw_materials i 9)
f-10    (icom-tuple Market_Floppies cash i 10.99999999)
f-11    (icom-tuple Market_Floppies labor_force m 10.99999999)
f-12    (icom-tuple Market_Floppies machines m 11.99999999)
f-13    (icom-tuple Buy_Floppies cash i 12.99999999)
f-14    (icom-tuple Buy_Floppies floppies o 13.99999999)
f-15    (icom-tuple Sell_in_Store_A floppies i 13.99999999)
f-16    (icom-tuple Sell_in_Store_A profit o 14.99999999)
f-17    (icom-tuple Manufacture_Floppies raw_materials i 15.99999999)
f-18    (icom-tuple Manufacture_Floppies quality_standards c 16.99999999)
f-19    (icom-tuple Manufacture_Floppies plans c 17.99999999)
f-20    (icom-tuple Manufacture_Floppies labor_force m 18.99999999)
f-21    (icom-tuple Manufacture_Floppies machines m 19.99999999)
f-22    (icom-tuple Manufacture_Floppies floppies o 20.99999999)
f-23    (icom-tuple Sell_in_Store_B floppies i 20.99999999)
f-24    (icom-activity-inputs Market_Floppies 2)
f-25    (icom-activity-controls Market_Floppies 4)
f-26    (icom-activity-outputs Market_Floppies 1)
f-27    (icom-activity-mechanisms Market_Floppies 2)
f-28    (icom-activity-inputs Buy_Floppies 1)
f-29    (icom-activity-controls Buy_Floppies 1)
f-30    (icom-activity-outputs Buy_Floppies 1)
f-31    (icom-activity-mechanisms Buy_Floppies 0)
f-32    (icom-activity-inputs Sell_in_Store_A 1)
```

```
f-33    (icom-activity-controls Sell_in_Store_A 1)
f-34    (icom-activity-outputs Sell_in_Store_A 1)
f-35    (icom-activity-mechanisms Sell_in_Store_A 0)
f-36    (icom-activity-inputs Manufacture_Floppies 1)
f-37    (icom-activity-controls Manufacture_Floppies 3)
f-38    (icom-activity-outputs Manufacture_Floppies 1)
f-39    (icom-activity-mechanisms Manufacture_Floppies 2)
f-40    (icom-activity-inputs Sell_in_Store_B 1)
f-41    (icom-activity-controls Sell_in_Store_B 0)
f-42    (icom-activity-outputs Sell_in_Store_B 0)
f-43    (icom-activity-mechanisms Sell_in_Store_B 0)
f-44    (project-name Market_Floppies)
f-45    (act-name Market_Floppies)
f-46    (act-numb Market_Floppies A0)
f-47    (act-desc Market_Floppies not-null)
f-48    (act-has-child Market_Floppies Buy_Floppies)
f-49    (act-has-child Market_Floppies Sell_in_Store_A)
f-50    (act-has-child Market_Floppies Manufacture_Floppies)
f-51    (act-has-child Market_Floppies Sell_in_Store_B)
f-52    (act-name Buy_Floppies)
f-53    (act-numb Buy_Floppies A1)
f-54    (act-desc Buy_Floppies not-null)
f-55    (act-has-child Buy_Floppies null)
f-56    (act-name Sell_in_Store_A)
f-57    (act-numb Sell_in_Store_A A2)
f-58    (act-desc Sell_in_Store_A not-null)
f-59    (act-has-child Sell_in_Store_A null)
f-60    (act-name Manufacture_Floppies)
f-61    (act-numb Manufacture_Floppies A3)
f-62    (act-desc Manufacture_Floppies not-null)
f-63    (act-has-child Manufacture_Floppies null)
f-64    (act-name Sell_in_Store_B)
f-65    (act-numb Sell_in_Store_B null)
f-66    (act-desc Sell_in_Store_B null)
f-67    (act-has-child Sell_in_Store_B null)
**********End of Working Memory********.

PRESS ANY KEY - THEN RETURN TO CONTINUE:

 Enter To select this operation
   1.  Assert all facts into the CLIPS Working Memory.
   2.  Display all the facts in CLIPS Working Memory.
   3.  Clear the CLIPS Working Memory.
   0.  EXIT
SELECT A NUMBER: 0

PRESS ANY KEY - THEN RETURN TO CONTINUE:

**************************************************
*  ESSENTIAL SUBSYSTEM EDITING AND DEBUGGING MENU   *
* Warning: These operations allow you to directly   *
```

```
* exercise the object operations.  Use extreme care.*
* --Essential Model and Utility Level Operations--  *
*****************************************************
 Enter     To select the desired submenu of operations
   1.      Activity Operations Menu
   2.      Data Element Operations Menu
   3.      Historical Activity Operations Menu
   4.      Calls Relation Operations Menu
   5.      ICOM Relation Operations Menu
   6.      Consists_Of Relation Operations Menu
   7       CLIPS Operations Menu
   8.      ICOM Fact Operations Menu
   9.      Activity Fact Operations Menu
   0.      EXIT
SELECT A NUMBER: 0

PRESS ANY KEY - THEN RETURN TO CONTINUE:

 *****************************************************
 *              THE ESSENTIAL SUBSYSTEM              *
 *          TEST AND DEMONSTRATION MAIN MENU         *
 *           -- SAtool II Level Operations --        *
 *****************************************************
 Enter     To select the desired operation
   1.      Restore (load) a project from disk
           (Warning: all current data cleared)
   2.      Save the current project to disk
   3.      Display the current project name
   4.      Change the current project name
   5.      Create and display a data dictionary entry
   6.      Add a box/activity to the project
   7.      Connect 2 boxes with a data element/arrow
   8.      Check Syntax of current project
   9.      -- Submenus for Low Level Operations --
   0.      EXIT
SELECT A NUMBER: 8
ICOM facts for CLIPS retrieved, if any.
CLIPS WM - a set of facts were asserted.
Project name fact retrieved.
CLIPS WM - a set of facts were asserted.
Activity facts for CLIPS retrieved, if any.
CLIPS WM - a set of facts were asserted.

****Essential Subsystem Syntax Messages****
WARNING: Activity Sell_in_Store_B needs a description.
ERROR: Activity Sell_in_Store_B must be numbered.
ERROR: Activity Sell_in_Store_B needs at least 1 output.
ERROR: Activity Sell_in_Store_B needs at least 1 control.

Clips run completed.  Rules fired =          6
```

```
PRESS ANY KEY - THEN RETURN TO CONTINUE:

 ******************************************************
 *              THE ESSENTIAL SUBSYSTEM               *
 *          TEST AND DEMONSTRATION MAIN MENU          *
 *           -- SAtool II Level Operations --         *
 ******************************************************
 Enter     To select the desired operation
   1.      Restore (load) a project from disk
           (Warning: all current data cleared)
   2.      Save the current project to disk
   3.      Display the current project name
   4.      Change the current project name
   5.      Create and display a data dictionary entry
   6.      Add a box/activity to the project
   7.      Connect 2 boxes with a data element/arrow
   8.      Check Syntax of current project
   9.      -- Submenus for Low Level Operations --
   0.      EXIT
SELECT A NUMBER: 0
csh> exit
csh>
script done on Sat Nov 17 20:51:12 1990
```

Bibliography

1. Aho, Alfred V. and others. *Data Structures and Algorithms*. Reading MA: Addison-Wesley Publishing Company, 1983.
2. Allen, Bradley P. and S. Daniel Lee. "Deploying Expert Systems in Ada." In *Proceedings of TRI-Ada'89*, pages 181–190, New York: ACM, Inc., 1989.
3. Austin, Kenneth A. and others. "An Entity-Relationship Modeling Approach to $IDEF_0$ Syntax," *Proceedings of IEEE 1990 National Aerospace and Electronics Conference NAECON 1990*, *2*:641–645 (May 1990).
4. Baker, Louis. *Artificial Intelligence With Ada*. New York: McGraw-Hill Publishing Company, Inc., 1989.
5. Booch, Grady. *Software Components With Ada: Structures, Tools and Subsystems*. Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.
6. Booch, Grady. *Software Engineering With Ada: Second Edition*. Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.
7. Booch, Grady. *Object Oriented Design With Applications*. Redwood City CA: Benjamin/Cummings Publishing Company, Inc., 1991.
8. Bowles, Adrion J. "A Note on the Yourdon Structured Method," *ACM Software Engineering Notes*, *15*(2):27 (April 1990).
9. Buchanan, Bruce G. and Reid G. Smith. "Fundamentals of Expert Systems." In Barr, Avron and others, editors, *The Handbook of Artificial Intelligence*, Volume 4, chapter 18, Reading MA: Addison-Wesley Publishing Company, 1989.
10. Chen, P. Pin-Shan. "The Entity-Relationship Model–Toward a Unified View of Data," *ACM Transactions on Database Systems*, *1*(1):9–36 (1976).
11. Connally, Ted D. *Common Database Interface for Heterogeneous Software Engineering Tools*. MS thesis, AFIT/GCS/ENG/87D-8, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A189628).
12. Department of Defense. *Computer Programming Language Policy*. DOD Directive 3405.1. Washington: Government Printing Office, 2 April 1987.
13. Department of Defense. *Use of Ada in Weapon Systems*. DOD Directive 3405.2. Washington: Government Printing Office, 30 March 1987.
14. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. PhD dissertation, University of Toronto, Toronto, Canada, 1984.
15. Hartrum, Thomas C. "$IDEF_0$ Requirements Analysis." Class handout for EENG 593, Systems and Software Analysis, October 30 1989.
16. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards* (Draft 4 Edition). Department of Electrical and Computer Engineering, Air Force Institute of Technology, January 2 1989.
17. Hayes-Roth, Frederick. "Rule-Based Systems," *Communications of the ACM*, *28*(9):921–932 (September 1985).
18. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures in Pascal: Second Edition*. Rockville MD: Computer Science Press, Inc., 1987.

19. Johnson, Steven E. *A Graphics Editor for Structured Analysis with a Data Dictionary*. MS thesis, AFIT/GE/ENG/87D-128, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A190618).

20. Jung, Donghak H. *Design of a Syntax Validation Tool for Requirements Analysis Using Structured Analysis and Design Technique(SADT)*. MS thesis, AFIT/GCS/ENG/88S-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1988 (AD-A202725).

21. Kameny, Iris and others. *Guide for the Management of Expert Systems Development*. Research Note R-3766-P&L, Santa Montica CA: National Defense Research Institute, Rand Corp, July 1989. Prepared under Contract MDA903-85-C-0030 for the Assistant Secretary of Defense for Production and Logistics.

22. Kameny, Iris and others. *Guide for the Management of Expert Systems Development: Additional Appendixes*. Research Note N-2970-P&L, Santa Montica CA: National Defense Research Institute, Rand Corp, August 1989. Prepared under Contract MDA903-85-C-0030 for the Assistant Secretary of Defense for Production and Logistics.

23. Kiem, Eric. "The Keystone System Design Methodology," *ACM Ada Letters*, *9*(5):101–108 (July/August 1989).

24. Kim, Intaek. *Expert System in Software Engineering Using Structured Analysis and Design Technique(SADT)*. MS thesis, AFIT/GCS/ENG/90J-2, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1990 (AD-A223022).

25. Kruchten, Philippe. "Error Handling in Large, Object-Based Ada Systems," *ACM Ada Letters*, *10*(7):91–103 (September/October 1990).

26. Lamont, Gary B. "An Introduction to Big-O and His Friends." Class handout for EENG 586, Advanced Information Structures, Fall Quarter 1988.

27. Luger, George F. and William A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems*. Redwood City CA: Benjamin/Cummings Publishing Company, Inc., 1989.

28. Marca, David A. and Clement L. McGowan. *SADT Structured Analysis and Design Technique*. New York: McGraw-Hill Book Company, 1988.

29. Martin, James and Steven Oxman. *Building Expert Systems: A Tutorial*. Englewood Cliffs NJ: McGraw-Hill Book Company, 1988.

30. Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH 45433. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF$_0$)*, June 1981. Contract F33615-78-C-5158 with SofTech, Inc.

31. Morris, Gerald R. *A Comparison of a Relational and Nested-Relational IDEF$_0$ Data Model*. MS thesis, AFIT/GCE/ENG/90M-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990 (AD-A220147).

32. NASA - Johnson Space Center. *CLIPS/Ada Advanced Programming Guide Version 4.3*, November 1989. Contract NAS9-18002 with Barrios Technology, Inc.

33. NASA - Johnson Space Center. *CLIPS/Ada Architecture Manual Version 4.3*, November 1989. Contract NAS9-18002 with Barrios Technology, Inc.

34. NASA - Johnson Space Center - Artificial Intelligence Section. *CLIPS Reference Manual: Version 4.3 of CLIPS*, July 1989.

35. NASA - Johnson Space Center - Artificial Intelligence Section. *CLIPS User's Guide: Version 4.3 of CLIPS*, August 1989. Written by Joseph C. Giarratano.

36. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, *SE-3*(1):16–34 (January 1977).

37. Shlaer, Sally and Steven J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs NJ: Prentice-Hall Inc, 1988.

38. Smith, Nealon F. *SAtool II: An $IDEF_0$ Syntax Data Manipulator and Graphics Editor*. MS thesis, AFIT/GCE/ENG/89D-8, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215289).

39. Sommerville, Ian. *Software Engineering: Third Edition*. Reading MA: Addison-Wesley Publishing Company, 1989.

40. Tevis, Jay-Evan J. *An Ada-based Framework for an $IDEF_0$ CASE Tool Using the X Window System*. MS thesis, AFIT/GCS/ENG/90D-15, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

41. Tsai, Jeffrey J. P. and J. C. Ridge. "Intelligent Support for Specification Transformation," *IEEE Software*, *3*(6):28–35 (December 1988).

42. Visible Systems Corporation. *The Visible Analyst Workbench*. Unnumbered Product Brochure. Waltham MA, 1989.

43. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs NJ: Prentice-Hall, Inc, 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
AN OBJECT ORIENTED DESIGN AND IMPLEMENTATION OF THE IDEF$_0$ ESSENTIAL DATA MODEL USING ADA AND AN ADA BASED EXPERT SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Terry L. Kitchen, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/90D-07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This investigation develops and implements an Ada based object oriented design (OOD) of the IDEF$_0$ Essential Data Model called the Essential Subsystem, which includes an Ada based expert system for IDEF$_0$ model syntax checking. IDEF$_0$ is a graphic approach to system description developed by SofTech, Inc. for the U.S. Air Force Program for Integrated Computer-Aided Manufacturing (ICAM) and is a subset of the Structured Analysis language. The IDEF$_0$ Essential Data Model is an entity-relationship (E-R) model of the IDEF$_0$ language and represents the fundamental (essential) information of an IDEF$_0$ model. The Essential Subsystem is so named because of its use as a subsystem with the Ada based, IDEF$_0$ CASE tool, SAtool II, that is under concurrent development. The development of SAtool II is part of ongoing research at the Air Force Institute of Technology (AFIT), with the Strategic Defense Initiative Organization (SDIO), on the use of IDEF$_0$ as a software requirements modeling methodology. The design phase includes the mapping of E-R constructs into objects in an OOD and the modeling of an expert system as just another object. The OOD is then implemented in Ada, including the expert system, which is implemented using CLIPS/Ada.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Engineering, Ada Programming Language, Expert Systems, ICAM Definition Method Zero (IDEF$_0$), Structured Analysis and Design Technique (SADT), Object Oriented Design, Entity Relationship Modeling	172
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL